VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA

Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112
(Approved by AICTE New Delhi, Permanently Affiliated to JNTUA, Ananthapuramu,
Accredited by NAAC, Recognized Under 2(F) &12(B) of UGC Act, An ISO 9001:2015 Certified Institute)

Department of CSE

Jawaharlal Nehru Technological University Anantapur

(Established by Govt. of A.P., Act. No. 30 of 2008)

Ananthapuramu–515 002 (A.P) India

II Year B.Tech R19 Regulations

DATA BASE MANAGEMENT SYSTEMS

UNIT – I

UNIT – I: Introduction: Database System Applications, Purpose of Database Systems, View of Data, Database Languages, Relational Databases, Database Design, Object-based and Semi-structured Databases, Data storage and Querying, Transaction Management, Mining and Analysis, Database Architecture, Database Users and Administrators.

Database Design and the E-R Model: Overview of the Design Process, The Entity-Relationship Model, Constraints, Entity-Relationship Diagrams, Entity – Relationship Design Issues, Weak Entity Sets, Extended E-R Features, Database Design for Banking Enterprise, Reduction to Relational Schemas, Other Aspects of Database Design

INTRODUCTION

A database management system (DBMS) is a collection of interrelated and a set of application programs used to access, update and manage that data. DBMS is also simply called as database system (DBS). This interrelated data is usually referred to as the database (DB).

- ❖ The goal of DBMS is to provide an environment that is both convenient and efficient to use in Retrieving information from the database, Storing information into the database.

Databases are usually designed to manage large bodies of information. This involves definition of structures for information storage (data modeling), provision of mechanisms for the manipulation of information (systems structure, query processing), providing for the safety of information in the database (crash recovery and security) and concurrency control if the system is shared by users.

DATABASE SYSTEM APPLICATIONS

- ✓ Banking: All Transactions
- ✓ Universities: Registration, Grades
- ✓ Sales: Customers, Products, Purchases
- ✓ Online Retailers: Order Tracking, Customized Recommendations
- ✓ Manufacturing: Production, Inventory, Orders, Supply Chain
- ✓ Airlines: Reservations, Schedules
- ✓ Human Resources: Employee Records, Salaries, Tax Deductions

- ❖ Databases touch all aspects of our lives
- ❖

PURPOSE OF DATABASE SYSTEMS

The typical file processing system is supported by a conventional operating system. The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files.

A file processing system has a number of major disadvantages.

1. **Data Redundancy and Inconsistency:** In file processing, every user group maintains its own files for handling its data processing applications. For example, consider the UNIVERSITY database. Here, two groups of users might be the course registration personnel and the

accounting office. The accounting office also keeps data on registration and related billing information, whereas the registration office keeps track of student courses and grades. Storing the same data multiple times is called data redundancy. This redundancy leads to several problems:

- ✓ Need to perform a single logical update multiple times.
- ✓ Storage space is wasted.
- ✓ Files that represent the same data may become inconsistent.
- ✓ Data inconsistency is the various copies of the same data may no larger agree.
  Example: One user group may enter a student's birth date erroneously as JAN-19-1984, whereas the other user groups may enter the correct value of JAN-29-1984.

2. **Difficulty in Accessing Data:** File processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Suppose that one of the bank officers needs to find out the names of all customers who live within a particular area. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. A program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.
3. **Data Isolation:** Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
4. **Integrity Problems:** The data values stored in the database must satisfy certain types of consistency constraints. Example: The balance of certain types of bank accounts may never fall below a prescribed amount. Developers enforce these constraints in the system by addition appropriate code in the various application programs
5. **Atomicity Problems:** Atomic means the transaction must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file processing system. Example: Consider a program to transfer $50 from account A to account B. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account A but was not credited to account B, resulting in an inconsistent database state.
6. **Concurrent Access Anomalies:** For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates is possible and may result in inconsistent data. To guard against this possibility, the system must maintain some form of supervision. But supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously. Example: When several reservation clerks try to assign a seat on an airline flight, the system should ensure that each seat can be accessed by only one clerk at a time for assignment to a passenger.
7. **Security Problems:** Enforcing security constraints to the file processing system is difficult.

VIEW OF DATA

Data Abstraction: A major purpose of a database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored and maintained. For the system to be usable, it must retrieve data efficiently. There are several levels of abstraction:

1. **Physical Level:** The lowest level of abstraction describes ***how*** the data are actually stored. The physical level describes complex low-level data structures in detail. Ex: index, B-tree, hashing.
2. **Logical Level:** The next-higher level of abstraction describes ***what*** data are stored in the database, and what relationships exist among those data. The logical level thus describes the entire database in terms of a small number of relatively simple structures. Database administrators, who must decide what information to keep in the database, use the logical level of abstraction.
3. **View Level:** Highest level of abstraction describes part of entire database for a particular group of users. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of the database system do not need all this information; instead, they need to access only a part of the database. View level of abstraction exists to simplify their interaction with the system. *The system may provide many views for same database.*

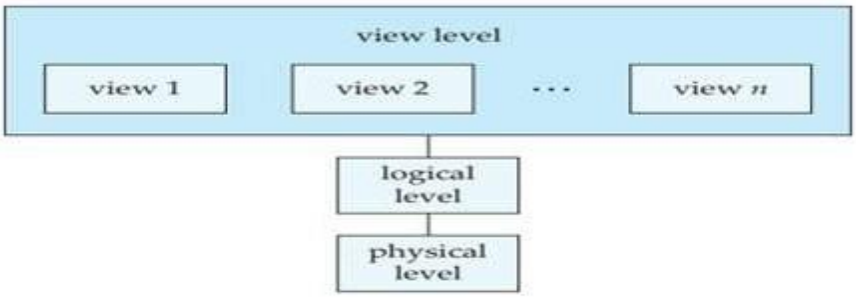


Figure The three levels of data abstraction.

Instances and Schemas: Similar to types and variables in programming languages.

Schema: The logical structure of the database (or) the overall design of the database is called the database schema. Schemas are changed infrequently, if at all. A database schema corresponds to the variable declarations (along with associated type definitions) in a program. Types of Schema include:

Physical Schema: describes the database design at the physical level

Logical Schema: describes the database design at the logical level.

A database may also have several schemas at the view level, sometimes called ***subschemas*** that describe different views of the database.

- The logical schema is the most important, in terms of its effect on application programs, since programmers construct applications by using the logical schema.

Instance: Collection of information stored in the database at a particular moment is called an instance of the database. Values of variables in a program at a point in time relate to an instance of database schema.

Physical Data Independence: The ability to modify the physical schema without changing the logical schema is known as Physical data independence. Usually, applications depend on the logical schema.

Logical Data Independence: Ability to modify logical schema without changing physical schema. It is harder to achieve as application programs are usually heavily dependent on logical structure of data.

DATA MODELS

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

The data models can be classified into four different categories:

1. **Relational Model:** The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Tables are also known as ***relations***. The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. *The collection of attributes and records will create the table*. The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model.
2. **Entity-Relationship Model:** The entity-relationship (E-R) data model uses a collection of basic objects, called ***entities***, and relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. The entity-relationship model is widely used in database design.
3. **Object-Based Data Model:** Object-oriented programming (especially in Java, C++, or C#) has become the dominant software-development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model. The object-relational data model combines features of the object-oriented data model and relational data model.
4. **Semi structured Data Model:** The semi structured data model permits the specification of data where individual data items of the same type may have different sets of attributes. The Extensible Markup Language (XML) is widely used to represent semi structured data.

- Historically, ***network data model*** and ***hierarchical data model*** preceded relational data model. These models were tied closely to the underlying implementation, and complicated task of modeling data. As a result they are used little now, except in old database code that is still in service in some places.

DATABASE LANGUAGES

A database system provides a ***Data-Definition Language*** to specify the database schema and a ***Data Manipulation Language*** to express database queries and updates. In practice, the data-definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

Data-Definition Language: "We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL)". DDL is also used to specify additional properties of data.

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

Example: Create Table account (account_number char (10), balance integer)

The data values stored in the database must satisfy certain consistency constraints. For example, suppose the university requires that the account balance of a department must never be negative. DDL provides facilities to specify such constraints. Database systems implement ***integrity constraints*** that can be tested:

- ***Domain Constraints:*** A domain of possible values must be associated with every attribute (for example, integer types, character types, date/time types). Declaring an attribute to be of a particular domain acts as a constraint on the values that it can take. These are the most elementary form of integrity constraint.
- ***Referential Integrity:***. There are cases where we wish to ensure that a value that appears in one relation for a given set of attributes also appears in a certain set of attributes in another relation (referential integrity). For example, the dept_name value in a course record must appear in the dept_name attribute of some record of the department relation. Database modifications can cause violations of referential integrity. When a referential integrity constraint is violated, normal procedure is to reject the action that caused the violation.
- ***Assertions:*** An assertion is *any condition that the database must always satisfy*. Domain constraints and referential integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, "Every department must have at least five courses offered every semester" must be expressed as an assertion.
- ***Authorization:*** We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of authorization. (The most common authorization is: *read authorization*, which allows reading, but not modification of data; *insert authorization*, which allows insertion of new data, but not modification of existing data; *update authorization*, which allows modification, but not deletion, of data; and *delete authorization*, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization).

❖ We specify the storage structure and access methods used by the database system by a set of statements in a special type of DDL called a *data storage and definition language*. These statements define the implementation details of the database schemas, which are usually hidden from the users.

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

- ✓ Retrieval of information stored in the database (Select)
- ✓ Insertion of new information into the database (Insert)
- ✓ Deletion of information from the database (Delete)
- ✓ Modification of information stored in the database (Update)

There are basically two types of DMLs:

1. ***Procedural DMLs*** require a user to specify what data are needed and how to get those data.
2. ***Declarative DMLs*** (also referred to as nonprocedural DMLs) require a user to specify what data are needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than are procedural DMLs.

A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language. Although technically incorrect, it is common practice to use the terms query language and data-manipulation language synonymously. SQL is the most widely used query language. Levels of abstraction apply both to defining data, manipulating data.

RELATIONAL DATABASES

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

Tables: Each table has multiple columns and each column has a unique name. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type. It is also possible to create schemas in the relational model that have problems such as unnecessarily duplicated information. Figure below presents a sample relational database table of university instructors.

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

The *instructor* table

The instructor table shows, for example, that an instructor named Einstein with ID 22222 is a member of the Physics department and has an annual salary of $95,000. ***(Draw and Explain any table of your wish)***

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

Create Table department (dept_name char (20), building char (15), budget numeric (12,2));

Execution of the above DDL statement creates the department table with three columns: dept_name, building, and budget, each of which has a specific data type associated with it.

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

```
Select name
From instructor
Where dept_name = ‘History’;
```

The query specifies that those rows from the table instructor where the dept_name is History must be retrieved, and the name attribute of these rows must be displayed. If the query is run on table in previous figure, result will consist of two rows, one with the name El Said and the other with the name Califieri.

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

The *department* table

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

```
Select instructor.ID, department.dept_name
From instructor, department
Where instructor.dept_name= department.dept_name and department.budget > 95000;
```

If above query were run on the tables in above figure, the system would find that there are two departments with budget of greater than $95,000—Computer Science and Finance; there are five instructors in these departments. Thus, the result will consist of a table with two columns (ID, dept_name) and five rows: (12121, Finance), (45565, Computer Science), (10101, Computer Science), (83821, Computer Science), and (76543, Finance).

Database Access from Application Programs: SQL does not support actions such as input from users, output to displays, or communication over the network. Such computations and actions must be written in a host language, such as C, C++, or Java, with embedded SQL queries that access the data in the database. There are two ways to do this:

- By providing an application program interface (set of procedures) that can be used to send DML and DDL statements to the database and retrieve the results. The Open Database Connectivity (ODBC) standard for use with the C language is a commonly used application program interface standard. The Java Database Connectivity (JDBC) standard provides corresponding features to the Java language.
- By extending the host language syntax to embed DML calls within the host language program.

DATABASE DESIGN

Database systems are designed to manage large bodies of information. Database design mainly involves the design of the database schema. The design of a complete database application environment that meets the needs of the enterprise being modeled requires attention to a broader set of issues.

Design Process: A high-level data model provides the database designer with a conceptual framework in which to specify the data requirements of the database users, and how the database will be structured to fulfill these requirements.

- ✓ The initial phase of database design is to characterize fully the data needs of database users. The outcome of this phase is a specification of user requirements.
- ✓ Next, the designer chooses a data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database.
- ✓ The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another.

  The designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

- ❖ In terms of the relational model, the conceptual-design process involves decisions on what attributes we want to capture in the database and how to group these attributes to form the various tables. The "what" part is basically a business decision and the "how" part is mainly a computer-science problem. There are principally two ways to tackle the problem.
- ❖ First one is to use ***entity-relationship model***; the other is to employ a set of algorithms collectively known as ***normalization*** that takes as input the set of all attributes and generates a set of tables.
- ❖ The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases. In the **logical-design phase**, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used. The designer uses the resulting system-specific database schema in the subsequent **physical-design phase**, in which the physical features of the database are specified.

Database Design for a University Organization: The description that arises from this design phase serves as the basis for specifying the conceptual structure of the database. Here are the major characteristics of the university.

- ✓ University is organized into departments. Each department is identified by a unique name (dept_name).
- ✓ Each department has a list of courses it offers. Each course has associated with it a course_id, title, dept_name, and credits, and may also have have associated prerequisites.
- ✓ Instructors are identified by their unique ID. Each instructor has name, associated department (dept_name), and salary.
- ✓ Students are identified by their unique ID. Each student has a name, an associated major department (dept_name), and tot_cred (total credit hours the student earned thus far).
- ✓ The university maintains a list of classrooms, specifying the name of the building, room number, and room capacity.
- ✓ The university maintains a list of all classes (sections) taught. Each section is identified by a course_id, sec_id, year, and semester, and has associated with it a semester, year, building, room number, and time_slot_id.
- ✓ The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).

DATABASE DESIGN FOR UNIVERSITY

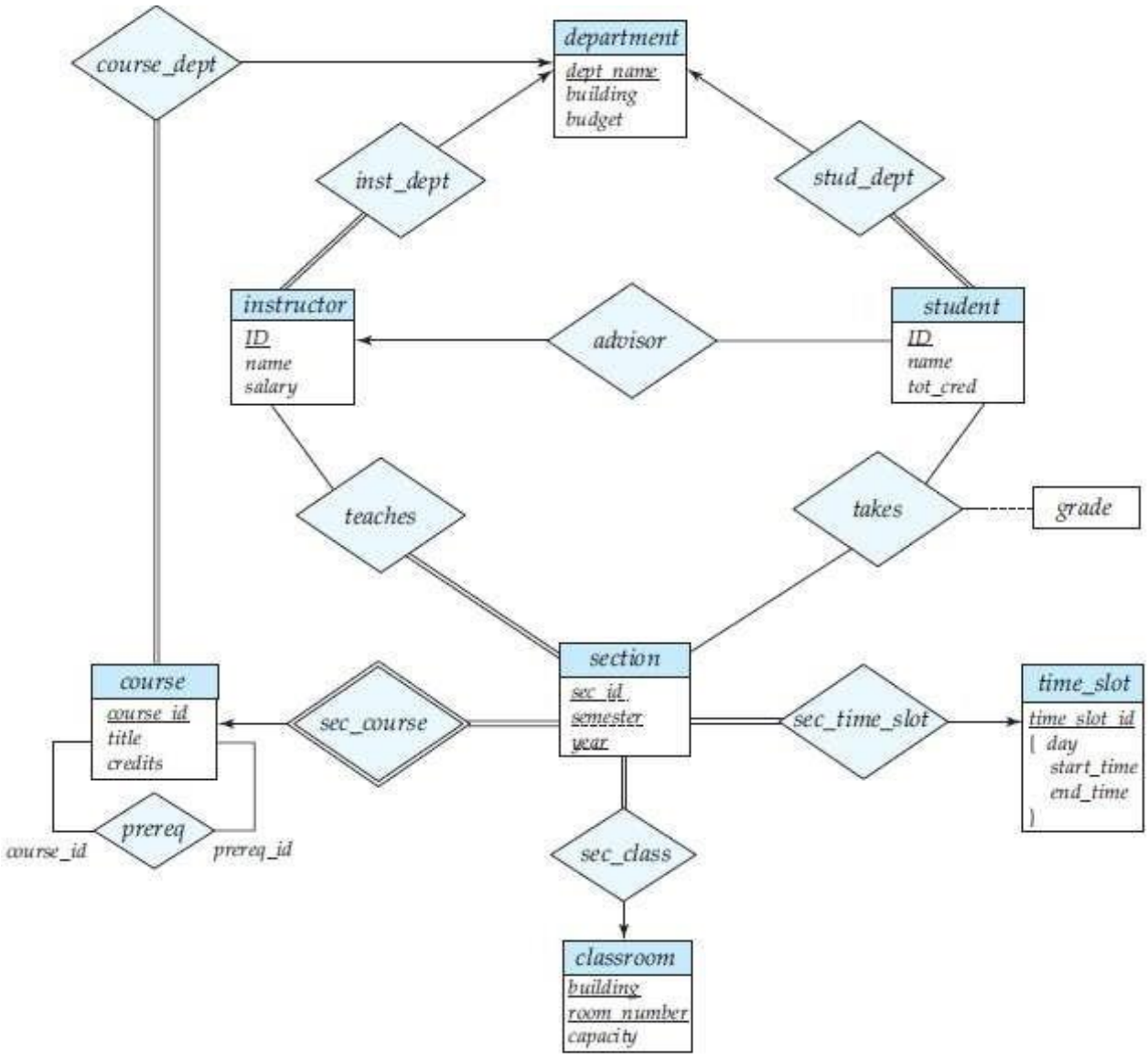


Figure E-R diagram for a university enterprise.

The Entity-Relationship Model: The entity-relationship (E-R) data model uses a collection of basic objects, called ***entities***, and ***relationships*** among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities. Entities are described in a database by a set of attributes. For example, the attributes ID, name, and salary may describe an instructor entity.

The extra attribute ID is used to identify an instructor uniquely (since it may be possible to have two instructors with the same name and the same salary). In the United States, many organizations use the social-security number of a person (a unique number the U.S. government assigns to every person in the United States) as a unique identifier.

- A relationship is an association among several entities. For example, a member relationship associates an instructor with her department. The set of all entities of the same type and the set of all relationships of the same type are termed an ***entity se***t and ***relationship set***, respectively.

The overall logical structure (schema) of a database can be expressed graphically by an entity-relationship (E-R) diagram. There are several ways in which to draw these diagrams. One of the most popular is to use the Unified Modeling Language (UML). In the notation we use, which is based on UML, an E-R diagram is represented as follows:

- ***Entity sets*** are represented by a partitioned rectangular box with the entity set name in the header and the attributes listed below it.
- ***Relationship sets*** are represented by a diamond connecting a pair of related entity sets. The name of the relationship is placed inside the diamond.

Fig: E-R Diagram

In addition to entities and relationships, the E-R model represents certain constraints to which the contents of a database must conform. One important constraint is mapping cardinalities, which express the number of entities to which another entity can be associated via a relationship set. For example, if each instructor must be associated with only a single department, the E-R model can express that constraint.

The entity-relationship model is widely used in database design.

Normalization: Another method for designing a relational database is to use normalization. To understand the need for normalization, let us look at what can go wrong in a bad database design. Among the undesirable properties that a bad design may have are:

- Repetition of information
- Inability to represent certain information

The goal of Normalization is to generate a set of relation schemas that allows us to store information without unnecessary redundancy, yet also allows us to retrieve information easily. The approach is to design schemas that are in an appropriate normal form. To determine whether a relation schema is in one of the desirable normal forms, we need additional information about the real-world enterprise that we are modeling with the database. The most common approach is to use functional dependencies.

DATABASE ARCHITECTURE

The architecture of a database system is greatly influenced by the underlying computer system on which the database system runs. Database systems can be centralized, or client-server, where one server machine executes work on behalf of multiple client machines. Database systems can also be designed to exploit parallel computer architectures. Distributed databases span multiple geographically separated machines.

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on which remote database users work, and server machines, on which the database system runs. The figure given provides a single picture of the various components of a database system and the connections among them.

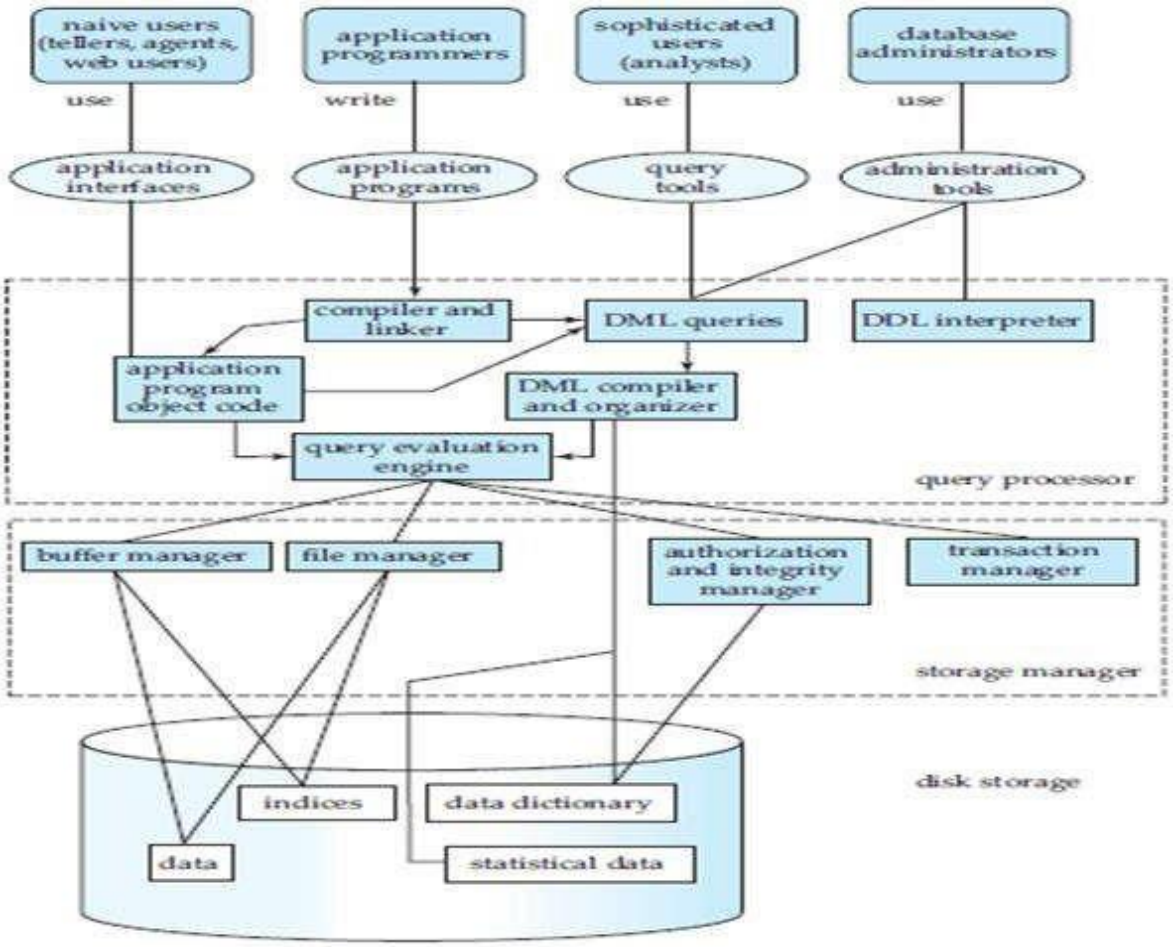


Figure System structure.

Database applications are usually partitioned into two or three parts. In two-tier architecture, the application resides at the client machine, where it invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

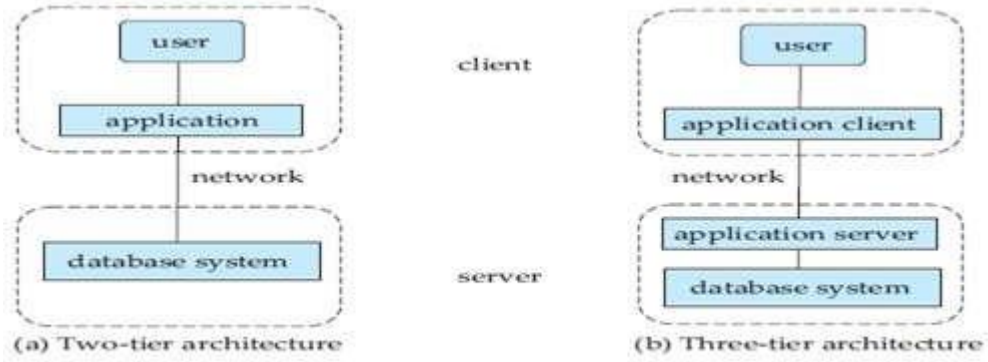


Figure Two-tier and three-tier architectures.

In contrast, in three-tier architecture, 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. Three-tier applications are more appropriate for large applications, and for applications that run on the WWW.

DATABASE USERS AND ADMINISTRATORS

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

Database Users and User Interfaces: There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

1) **Naïve Users:** Naïve users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. The typical user interface for naïve users is a forms interface, where the user can fill in appropriate fields of the form. Naïve users may also simply read reports generated from the database.

As an example, consider a student, who during class registration period, wishes to register for a class by using a Web interface. Such a user connects to a Web application program that runs at a Web server. The application first verifies the identity of the user, and allows her to access a form where she enters the desired information. The form information is sent back to the Web application at the server, which then determines if there is room in the class and if so adds the student information to the class roster in the database.

2) **Application Programmers:** Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. Rapid application development (RAD) tools are tools that enable an application programmer to construct forms and reports with minimal programming effort.
3) **Sophisticated Users:** Sophisticated users interact with the system without writing programs. Instead, they form their requests either using a database query language or by using tools such as data analysis software. Analysts who submit queries to explore data in the database fall in this category.
4) **Specialized Users:** Specialized users are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, 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 DBMS is to have central control of both the data and the programs that access those data. ***A person who has such central control over the system is called a database administrator (DBA)***. The **functions/responsibilities** of a DBA include:

1. **Schema Definition:** The DBA creates the original database schema by executing a set of data definition statements in the DDL.
2. **Storage Structure and Access-Method Definition**
3. **Schema and Physical-Organization Modification:** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

4. **Granting of Authorization for Data Access:** By granting different types of authorization, the database administrator can regulate which parts of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.
5. **Routine Maintenance:** Examples of the database administrator's routine maintenance activities are:
   - ✓ Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
   - ✓ Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
   - ✓ Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

DATA STORAGE AND QUERYING

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

Storage Manager: Storage manager is the component of a database system that provides interface between low-level data stored in database, application programs and queries submitted to the system.

The storage manager is important because databases typically require a large amount of storage space. Corporate databases range in size from hundreds of gigabytes to, for the largest databases, terabytes of data. Since the main memory of computers cannot store this much information, the information is stored on disks. Data are moved between disk storage and main memory as needed.

- ❖ *The storage manager is responsible for the interaction with the file manager*. The raw data are stored on the disk using the file system provided by the operating system. The storage manager is responsible for storing, retrieving, and updating data in the database.

The storage manager components include:

1. **Authorization and Integrity Manager**, which tests for the satisfaction of integrity constraints and checks the authority of users to access data.
2. **Transaction Manager**, which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.
3. **File Manager**, which manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
4. **Buffer Manager**, which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. Buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements ***several data structures*** as part of the physical system implementation:

- ✓ Data files, which store the database itself.
- ✓ Data dictionary, which stores metadata about structure of database, in particular schema of database.
- ✓ Indices, which can provide fast access to data items. Like the index in a textbook, a database index provides pointers to those data items that hold a particular value.

The Query Processor: The query processor is important because it helps the database system to simplify and facilitate access to data. It allows database users to obtain good performance while being able to work at the view level. It is the job of the database system to translate updates and queries written in a nonprocedural language, at the logical level, into an efficient sequence of operations at the physical level. The query processor components include:

1. **DDL interpreter**, which interprets DDL statements and records the definitions in the data dictionary.
2. **DML compiler**, which translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands. The DML compiler also performs *query optimization*; that is, it picks lowest cost evaluation plan among the alternatives.
3. **Query evaluation engine**, which executes low-level instructions generated by the DML compiler.

SPECIALITY (OBJECT-BASED AND SEMI-STRUCTURED) DATABASES

Object-Based Data Models: Object-oriented programming has become the dominant software development methodology. This led to the development of an object-oriented data model that can be seen as extending the E-R model with notions of encapsulation, methods (functions), and object identity. Inheritance, and encapsulation (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modeling. The object-oriented data model also supports a rich type system, including structured and collection types. In the 1980s, several database systems based on the object-oriented data model were developed. The major database vendors presently support the object-relational data model, a data model that combines features of the object-oriented data model and relational data model.

Semi Structured Data Models: Semi structured data models permit the specification of data where individual data items of the same type may have different sets of attributes. This is in contrast with data models mentioned earlier, where every data item of a particular type must have the same set of attributes.

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

TRANSACTION MANAGEMENT

Often, several operations on the database form a single logical unit of work. An example is a funds transfer, in which one department account (say A) is debited and another department account (say B) is credited. Clearly, it is essential that either both the credit and debit occur, or that neither occur. That is, the funds transfer must happen in its entirety or not at all. This all-or-none requirement is called ***atomicity***. In addition, it is essential that the execution of the funds transfer preserve the consistency of the database. That is, the value of the sum of the balances of A and B must be preserved. This correctness requirement is called ***consistency***. Finally, after the successful execution of a funds transfer, the new values of the balances of accounts A and B must persist, despite the possibility of system failure. This persistence requirement is called ***durability***.

"A transaction is a collection of operations that performs a single logical function in a database application". Each transaction is a unit of both atomicity and consistency. Thus, we require that transactions do not violate any database-consistency constraints. That is, if the database was consistent when a transaction started, the database must be consistent when the transaction successfully terminates. It is the responsibility of *programmer*.

Ensuring atomicity and durability properties is the responsibility of *recovery manager*. In the absence of failures, all transactions complete successfully, and atomicity is achieved easily. However, because of various types of failure, a transaction may not always complete its execution successfully. If we are to ensure the atomicity property, a failed transaction must have no effect on the state of the database. Thus, database must be restored to the state in which it was before the transaction in question started executing. The database system must therefore perform failure recovery, that is, detect system failures and restore the database to the state that existed prior to the occurrence of the failure.

Finally, when several transactions update the database concurrently, the consistency of data may no longer be preserved, even though each individual transaction is correct. It is the responsibility of the *concurrency-control manager* to control interaction among concurrent transactions, to ensure consistency of database. ***Transaction manager consists of concurrency-control manager and recovery manager***.

MINING AND ANALYSIS

The term data mining refers to the process of semi automatically analyzing large databases to find useful patterns. Data mining deals with large volumes of data, stored primarily on disk. That is, data mining deals with "knowledge discovery in databases." There are a variety of possible types of patterns that may be useful, and different techniques are used to find different types of patterns.

Usually there is a manual component to data mining, consisting of preprocessing data to a form acceptable to algorithms, post processing of discovered patterns to find novel ones that could be useful. There may also be more than one type of pattern that can be discovered from a given database, manual interaction may be needed to pick useful types of patterns. For this reason, data mining is really a semiautomatic process in real life.

Several techniques and tools are available to help with decision support. Several tools for data analysis allow analysts to view data in different ways. Other analysis tools pre compute summaries of very large amounts of data, to give fast responses to queries. The SQL standard contains additional constructs to support data analysis.

Large companies have diverse sources of data that they need to use for making business decisions. To execute queries efficiently on such diverse data, companies have built data warehouses. Data warehouses gather data from multiple sources under a unified schema, at a single site. Thus, they provide the user a single uniform interface to data.

Textual data is unstructured, unlike the rigidly structured data in relational databases. Querying of unstructured textual data is referred to as ***information retrieval***. Information retrieval systems have much in common with database systems—in particular, the storage and retrieval of data on secondary storage.

DATABASE DESIGN AND THE E-R MODEL

Overview of the Design Process: The task of creating a database application is a complex one, involving design of the database schema, design of the programs that access and update the data, and design of a security scheme to control access to data. ***Needs of users*** play a central role in design process.

Design Phases: For small applications, it may be feasible for a database designer who understands the application requirements to decide directly on the relations to be created, their attributes, and constraints on the relations. A high-level and complex data model provides database designer with a conceptual framework to specify data requirements of database users, and how database will be structured to fulfill these requirements.

- ✓ The initial phase of database design is to characterize fully the data needs of database users. The outcome of this phase is a specification of user requirements.
- ✓ Next, the designer chooses E-R data model, and by applying the concepts of the chosen data model, translates these requirements into a conceptual schema of the database.
- ✓ The schema developed at this conceptual-design phase provides a detailed overview of the enterprise. The entity-relationship model is typically used to represent the conceptual design. By specifying the entities, the attributes of the entities, the relationships among the entities, and constraints on the entities and relationships.

Typically, the conceptual-design phase results in the creation of an entity-relationship diagram that provides a graphic representation of the schema.

- The designer reviews the schema to confirm that all data requirements are indeed satisfied and are not in conflict with one another. Designer can also examine the design to remove any redundant features. The focus at this point is on describing the data and their relationships, rather than on specifying physical storage details.

The process of moving from an abstract data model to the implementation of the database proceeds in two final design phases.

- ❖ In the ***logical-design phase***, the designer maps the high-level conceptual schema onto the implementation data model of the database system that will be used.
- ❖ Finally, the designer uses the resulting system-specific database schema in the subsequent ***physical-design phase***, in which the physical features of the database are specified.

Note: The physical schema of a database can be changed relatively easily after an application has been built. However, changes to the logical schema are usually harder to carry out, since they may affect a number of queries and updates scattered across application code. It is therefore important to carry out the database design phase with care, before building the rest of the database application.

Design Alternatives: A major part of the database design process is deciding how to represent in the design the various types of "things" such as people, places, products, and the like. We use the term entity to refer to any such distinctly identifiable item. In a university database, examples of entities would include instructors, students, and departments. Various entities are related to each other in a variety of ways, all of which need to be captured in database design.

In designing a database schema, we must avoid two major pitfalls:

1. **Redundancy:** A bad design may repeat information. Redundancy can also occur in a relational schema. The biggest problem with redundant information is that the copies of a piece of information can become inconsistent if the information is updated without taking precautions to update all copies of the information. Ideally, information should appear in exactly one place.
2. **Incompleteness:** A bad design may make certain aspects of enterprise difficult/impossible to model. For example, suppose that, we only had entities corresponding to course offering, without having an entity corresponding to courses. Equivalently, in terms of relations, suppose we have a single relation where we repeat all of the course information once for each section that the course is offered. It would then be impossible to represent information about a new course, unless that course is offered.

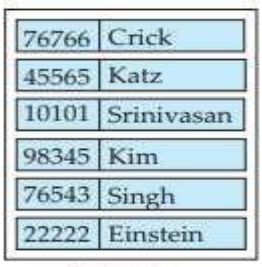
THE ENTITY-RELATIONSHIP MODEL

The entity-relationship (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database. The E-R model is very useful in mapping the meanings and interactions of real-world enterprises onto a conceptual schema. Because of this usefulness, many database-design tools draw on concepts from the E-R model. The E-R data model employs three basic concepts: entity sets, relationship sets, and attributes.

Entity Sets: An ***entity*** is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in a university is an entity. An entity has a set of properties, and the values for some set of properties may uniquely identify an entity. For instance, a person may have a person id property whose value uniquely identifies that person. An entity may be concrete, such as a person or a book, or it may be abstract, such as a course, a course offering, or a flight reservation.

An ***entity set*** is a set of entities of the same type that share the same properties, or attributes. The set of all people who are instructors at a given university, for example, can be defined as the entity set instructor. We use the term extension of the entity set to refer to the actual collection of entities belonging to the entity set. Thus, the set of actual instructors in the university forms the extension of the entity set instructor. The above distinction is similar to the difference between a relation and a relation instance.

An entity is represented by a set of ***attributes***. Attributes are descriptive properties possessed by each member of an entity set. Possible attributes of instructor entity set are ID, name, dept_name, and salary. Each entity has a ***value*** for each of its attributes. For instance, a particular instructor entity may have value 12121 for ID, the value Wu for name, the value Finance for dept_name, and the value 90000 for salary.



76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

Fig: Instructor Entity Set

Relationship Sets: A ***relationship*** is an association among several entities. A ***relationship set*** is a set of relationships of same type. Formally, it is a mathematical relation on n ≥ 2 (possibly non-distinct) entity sets.

If E1, E2,..., En are entity sets, then a relationship set R is a subset of

{(e1,e2,...,en) | e1 ∈ E1,e2 ∈ E2,...,en ∈ En} where (e1,e2,...,en) is a relationship.

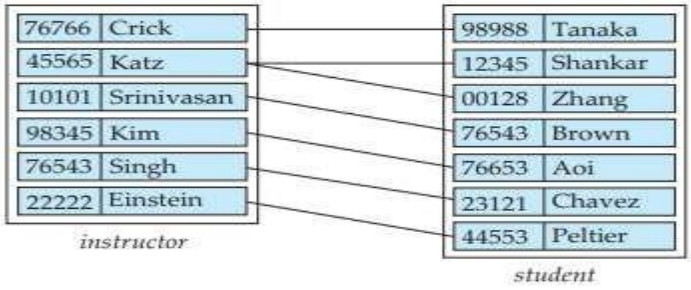


Figure Relationship set *advisor*.

Consider the two entity sets instructor and student in Figure. We define the relationship set advisor to denote the association between instructors and students. The association between entity sets is referred to as ***participation***; that is, the entity sets E1, E2,..., En participate in relationship set R. A relationship instance in an E-R schema represents an association between the named entities.

Ex: The individual instructor entity Katz, who has instructor ID 45565, and the student entity Shankar, who has student ID 12345, participate in a relationship instance of advisor. This relationship instance represents that in the university, the instructor Katz is advising student Shankar.

The function that an entity plays in a relationship is called that entity's ***role***. Since entity sets participating in a relationship set are generally distinct, roles are implicit and are not usually specified. However, they are useful when the meaning of a relationship needs clarification.

A relationship may also have attributes called ***descriptive attributes***. Consider a relationship set advisor with entity sets instructor and student. We could associate the attribute date with that relationship to specify the date when an instructor became the advisor of a student. Below figure shows the relationship set advisor with a descriptive attribute date.

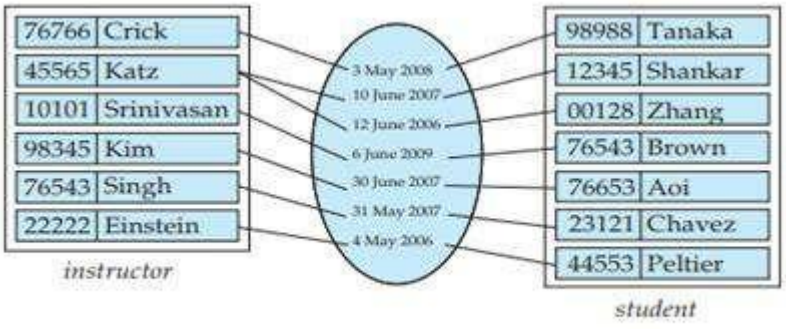


Figure *date* as attribute of the *advisor* relationship set.

The relationship set advisor provides an example of a binary relationship set— that is, one that involves two entity sets. Most of the relationship sets in a database system are binary. Occasionally, however, relationship sets involve more than two entity sets, often called as ternary relationships or non-binary or

n-ary relationships. The number of entity sets that participate in a relationship set is the ***degree*** of the relationship set. A binary relationship set is of degree 2; a ternary relationship set is of degree 3.

Attributes: For each attribute, there is a set of permitted values, called the ***domain***, or ***value set***. The domain attribute semester might be strings from the set {Fall, Winter, Spring, Summer}. Since an entity set may have several attributes, each entity can be described by a set of (attribute, data value) pairs.

Ex: A particular instructor entity may be described by the set {(ID, 76766), (name, Crick), (dept_name, Biology), (salary, 72000)}, meaning that the entity describes a person named Crick whose instructor ID is 76766, who is a member of the Biology department with salary of $72,000.

An attribute, as used in the E-R model, can be characterized by the following attribute types.

1. **Simple and Composite Attributes:** In our examples thus far, the attributes have been simple; that is, they have not been divided into subparts. Composite attributes, on the other hand, can be divided into subparts (that is, other attributes). For example, an attribute name could be structured as a composite attribute consisting of first name, middle name, and last name. Composite attributes help us to group together related attributes, making the modeling cleaner.

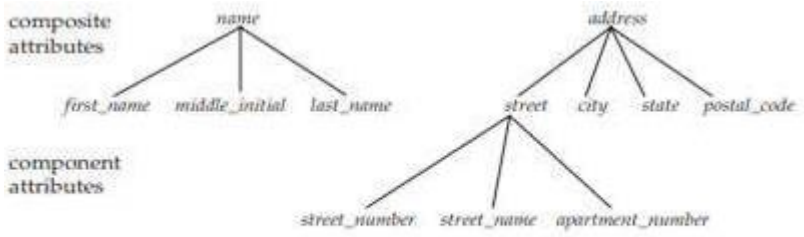


Figure Composite attributes instructor *name* and *address*.

Note: Using composite attributes in a design schema is a good choice if a user will wish to refer to an entire attribute on some occasions and to only a component of the attribute on other occasions.

2. **Single-valued and Multivalued Attributes:** There may be instances where an attribute has a set of values for a specific entity. Suppose an instructor may have one, or several phone numbers, and different instructors may have different numbers of phones. This type of attribute is said to be ***multivalued***. To denote that an attribute is multivalued, we enclose it in braces, for example {phone_number} or {dependent name}.

Note: Upper and lower bounds may be placed on number of values in a multivalued attribute, if needed.

3. **Derived Attributes:** Value for Derived attribute can be derived from values of other related attributes or entities. For instance, suppose that instructor entity set has an attribute age. If instructor entity set also has an attribute date_of_birth, we can calculate age from date_of_birth and current date. Thus, age is a derived attribute. In this case, date_of_birth may be referred to as a base attribute, or a stored attribute. Value of a derived attribute is not stored but is computed when required.

✓ An attribute takes a null value when an entity does not have a value for it. The null value may indicate "not applicable" (that is, that the value does not exist for the entity) or "unknown". An unknown value

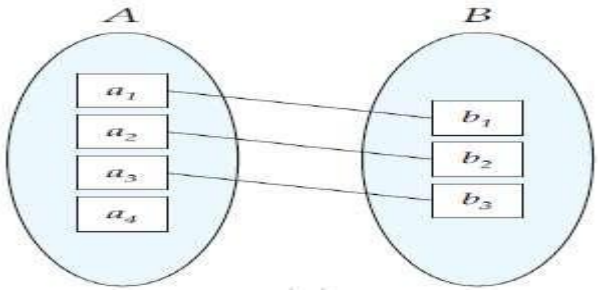
may be either missing (the value does exist, but we do not have that information) or not known (we do not know whether or not the value actually exists). A null value for the apartment number attribute could mean that the address does not include an apartment number (not applicable), that an apartment number exists but we do not know what it is (missing), or that we do not know whether or not an apartment number is part of the instructor's address (unknown).

CONSTRAINTS

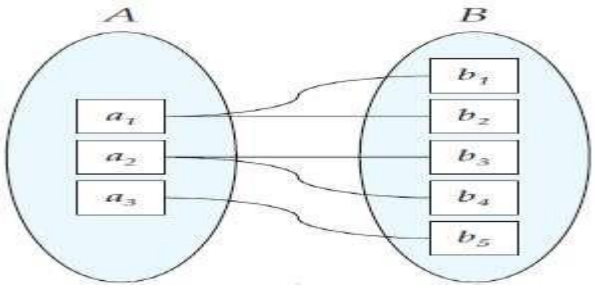
An E-R schema may define certain constraints to which the contents of a database must conform.

Mapping Cardinalities: Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set. Mapping cardinalities are most useful in describing binary relationship sets. For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

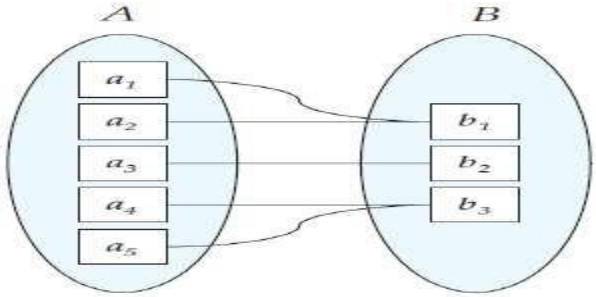
1) **One-to-One:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.



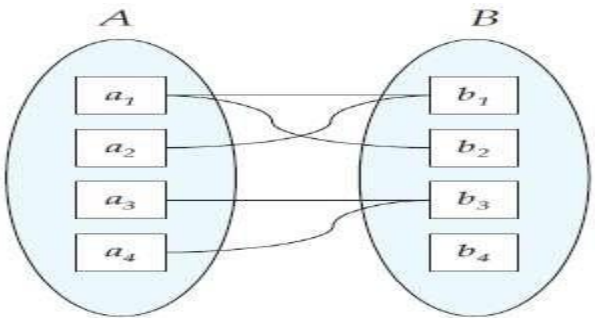
2) **One-to-Many:** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, can be associated with at most one entity in A.



3) **Many-to-One:** An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A.



4) **Many-to-Many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling. As an illustration, consider the advisor relationship set. If, in a particular university, a student can be advised by only one instructor, and an instructor can advise several students, then the relationship set from instructor to student is one-to-many. If a student can be advised by several instructors (as in case of students advised jointly), relationship set is many-to-many.

Participation Constraints: The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship in R. If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be partial.

Keys: We must have a way to specify how entities within a given entity set are distinguished. Conceptually, individual entities are distinct; from a database perspective, however, the differences among them must be expressed in terms of their attributes. Therefore, the values of the attribute values of an entity must be such that they can uniquely identify the entity. In other words, no two entities in an entity set are allowed to have exactly the same value for all attributes. ***A key for an entity is a set of attributes that suffice to distinguish entities from each other.*** Keys also help to identify relationships uniquely, and thus distinguish relationships from each other.

The **Primary Key** of an entity set allows us to distinguish among the various entities of the set. We use a similar mechanism to distinguish among the various relationships of a relationship set. The structure of the primary key for the relationship set depends on the mapping cardinality of the relationship set.

A ***super key*** is a set of columns (attributes) to uniquely identify rows in a table. If the attribute names of primary keys are not unique across entity sets, the attributes are renamed to distinguish them; the name of the entity set combined with the name of the attribute would form a unique name.

- If an entity set participates more than once in a relationship set, the role name is used instead of the name of the entity set, to form a unique attribute name.

REMOVING REDUNDANT ATTRIBUTES IN ENTITY SETS

When we design a database using the E-R model, we usually start by *identifying the entity sets* that should be included. For example, in the university organization, we include entity sets such as student, instructor, etc. Then we must *choose the appropriate attributes*. In the university organization, we decided that for the instructor entity set, we will include the attributes ID, name, dept_name, and salary. The choice of what attributes to include is up to the designer, who has a good understanding of the structure of the enterprise. Once the entities and their corresponding attributes are chosen, *the relationship sets among the various entities are formed*.

These relationship sets may result in a situation where attributes in the various entity sets are redundant and need to be removed from the original entity sets. To illustrate, consider the entity sets instructor and department:

- ❖ Entity set instructor includes attributes ID, name, dept_name, and salary, with ID forming primary key.
- ❖ Entity set department includes attributes dept_name, building, and budget, with dept_name forming primary key.

The attribute dept_name appears in both entity sets. Since it is the primary key for the entity set department, it is redundant in the entity set instructor and needs to be removed. When we create a relational schema from the E-R diagram, the attribute dept_name in fact gets added to the relation instructor, but only if each instructor has at most one associated department. If an instructor has more than one associated department, the relationship between instructors and departments is recorded in a separate relation inst_dept.

Note: Treating the connection between instructors and departments uniformly as a relationship, rather than as an attribute of instructor, makes the logical relationship explicit, and helps avoid a premature assumption that each instructor is associated with only one department.

A good entity-relationship design does not contain redundant attributes.

For our university example, we list entity sets and their attributes below, with primary keys underlined:

- ✓ classroom: with attributes (building, room_number, capacity).
- ✓ department: with attributes (dept_name, building, budget).
- ✓ course: with attributes (course_id, title, credits).
- ✓ instructor: with attributes (ID, name, salary).
- ✓ section: with attributes (course_id, sec_id, semester, year).
- ✓ student: with attributes (ID, name, tot_cred).

The relationship sets in our design are listed below:

- ✓ inst_dept: relating instructors with departments.
- ✓ stud_dept: relating students with departments.
- ✓ teaches: relating instructors with sections.
- ✓ takes: relating students with sections, with a descriptive attribute grade.
- ✓ course_dept: relating courses with departments.
- ✓ sec_course: relating sections with courses.
- ✓ sec_class: relating sections with classrooms.
- ✓ advisor: relating students with instructors.

You can verify that none of the entity sets has any attribute that is made redundant by one of the relationship sets.

ENTITY-RELATIONSHIP DIAGRAMS

An **E-R diagram** can express the overall logical structure of a database graphically. *E-R diagrams are simple and clear*—qualities that may well account in large part for the widespread use of the E-R model.

Basic Structure: An E-R diagram consists of the following major components:

- **Rectangles divided into two parts** represent entity sets. The first part, contains the name of the entity set. The second part contains the names of all the attributes of the entity set.
- **Diamonds** represent relationship sets.
- **Undivided rectangles** represent the attributes of a relationship set. Attributes that are part of the primary key are underlined.
- **Lines** link entity sets to relationship sets.
- **Dashed lines** link attributes of a relationship set to the relationship set.
- **Double lines** indicate total participation of an entity in a relationship set.
- **Double diamonds** represent identifying relationship sets linked to weak entity sets.

Figure E-R diagram corresponding to instructors and students.

Consider the E-R diagram in figure, which consists of two entity sets, *instructor* and *student* related through a binary relationship set *advisor*. The attributes associated with *instructor* are *ID*, *name*, and *salary*. The attributes associated with *student* are *ID*, *name*, and *tot_cred*. In figure, attributes of an entity set that are members of the primary key are underlined.

If a relationship set has some attributes associated with it, then we enclose the attributes in a rectangle and link the rectangle with a dashed line to the diamond representing that relationship set. For example, in below figure, we have the *date* descriptive attribute attached to the relationship set *advisor* to specify the date on which an instructor became the advisor.

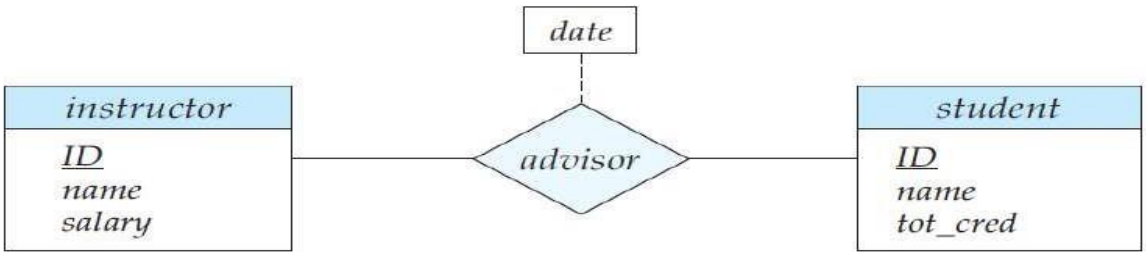


Figure E-R diagram with an attribute attached to a relationship set.

Mapping Cardinality: The relationship set *advisor*, between the *instructor* and *student* entity sets may be one-to-one, one-to-many, many-to-one, or many-to-many. To distinguish among these types, we draw either a directed line (→) or an undirected line (—) between the relationship set and the entity set in question, as follows:

1) One-to-one: We draw a directed line from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise at most one student, and a student may have at most one advisor.

2) One-to-many: We draw a directed line from the relationship set *advisor* to the entity set *instructor* and an undirected line to the entity set *student*. This indicates that an instructor may advise many students, but a student may have at most one advisor.

3) Many-to-one: We draw an undirected line from the relationship set *advisor* to the entity set *instructor* and a directed line to the entity set *student*. This indicates that an instructor may advise at most one student, but a student may have many advisors.

4) Many-to-many: We draw an undirected line from the relationship set *advisor* to both entity sets *instructor* and *student*. This indicates that an instructor may advise many students, and a student may have many advisors.

E-R diagrams also provide a way to indicate more complex constraints on the number of times each entity participates in relationships in a relationship set. ***A line may have an associated minimum and maximum cardinality, shown in the form l..h, where l is the minimum and h the maximum cardinality.*** A minimum value of 1 indicates total participation of the entity set in the relationship set; that is, each entity in the entity set occurs in at least one relationship in that relationship set. A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value ∗ indicates no limit.

Figure Cardinality limits on relationship sets.

For example, consider the figure, the line between *advisor* and *student* has a cardinality constraint of 1..1, meaning the minimum and the maximum cardinality are both 1. That is, each student must have exactly one advisor. The limit 0..∗ on the line between *advisor* and *instructor* indicates that an instructor can have

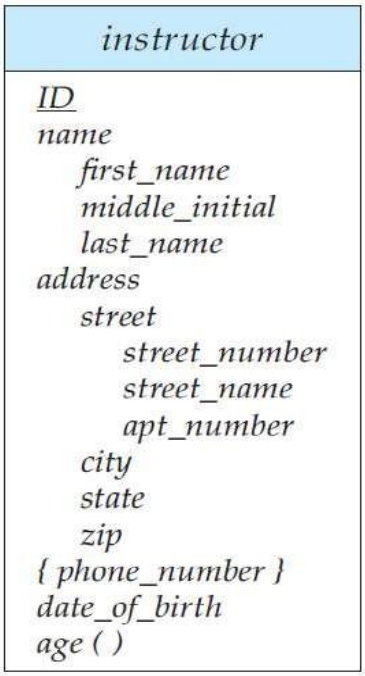
zero or more students. Thus, the relationship *advisor* is one-to-many from *instructor* to *student*, and further the participation of *student* in *advisor* is total, implying that a student must have an advisor.

If both edges have a maximum value of 1, relationship is one-to-one. If we had specified a cardinality limit of 1..∗ on left edge, we would be saying that each instructor must advise at least one student.

Note: The E-R diagram in the figure could alternatively have been drawn with a double line from *student* to *advisor* and an arrow on the line from *advisor* to *instructor*, in place of cardinality constraints shown. This alternative diagram would enforce exactly the same constraints as the constraints shown in figure.

Complex Attributes: Following figure shows how composite attributes can be represented in the E-R notation. As an example, suppose we were to add an address to the *instructor* entity-set. The address can be defined as the composite attribute *address* with the attributes: *street*, *city*, *state*, and *zip code*. The attribute *street* is itself a composite attribute whose component attributes are *street number*, *street name*, and *apartment number*.

The figure also illustrates a multivalued attribute *phone_number*, denoted by "*{phone_number}*", and a derived attribute *age*, depicted by a "*age* ( )".



Roles: We indicate roles in E-R diagrams by labeling the lines that connect diamonds to rectangles. Figure 7.12 shows the role indicators *course_id* and *prereq id* between the *course* entity set and the *prereq* relationship set.

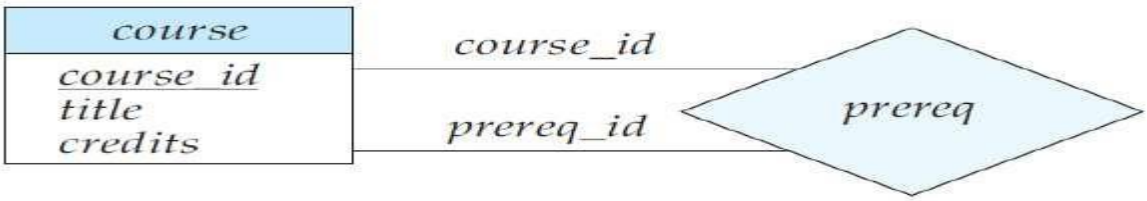


Figure E-R diagram with role indicators.

Non-binary/n-ary/Ternary Relationship Sets: Non-binary/n-ary/Ternary relationship sets can be specified easily in an E-R diagram. Figure 7.13 consists of the three entity sets *instructor*, *student*, and *project*, related through the relationship set *proj_guide*.

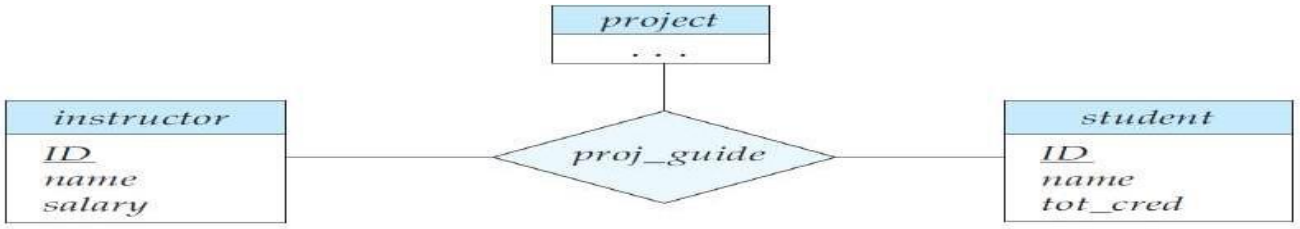


Figure E-R diagram with a ternary relationship.

We can specify some types of many-to-one relationships in the case of non-binary relationship sets. Suppose a *student* can have at most one instructor as a guide on a project. This constraint can be specified by an arrow pointing to *instructor* on the edge from *proj_guide*.

We permit at most one arrow out of a relationship set, since an E-R diagram with two or more arrows out of a non-binary relationship set can be interpreted in two ways.

Suppose there is a relationship set R between entity sets $A_1, A_2, \ldots, A_n$, and the only arrows are on the edges to entity sets $A_{i+1}, A_{i+2}, \ldots, A_n$. Then, the two possible interpretations are:

1. A particular combination of entities from $A_1, A_2, \ldots, A_i$ can be associated with at most one combination of entities from $A_{i+1}, A_{i+2}, \ldots, A_n$. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of $A_1, A_2, \ldots, A_i$.

2. For each entity set A_k, $i < k \leq n$, each combination of the entities from other entity sets can be associated with at most one entity from A_k. Each set $\{A_1, A_2, \ldots, A_{k-1}, A_{k+1}, \ldots, A_n\}$, for $i < k \leq n$, then forms a candidate key.

To avoid confusion, we permit only one arrow out of a relationship set, in which case the two interpretations are equivalent.

Weak Entity Sets: An entity set that is dependent on other entity set (or) an entity set that does not have a primary key is referred to as a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.

In our example, the identifying entity set for *section* is *course*, and the relationship *sec_course*, which associates *section* entities with their corresponding *course* entities, is the identifying relationship.

The **discriminator (***partial key***)** of a weak entity set is a set of attributes that allows the distinction to be made as weak entity set does not have Primary Key. For example, the discriminator of the weak

entity set *section* consists of the attributes *sec_id*, *year*, and *semester*, since, for each course, this set of attributes uniquely identifies one single section for that course.

The primary key of a weak entity set is formed by the primary key of the identifying entity set, plus the weak entity set's discriminator. In the case of the entity set *section*, its primary key is *{course_id, sec_id, year, semester}*, where *course_id* is the primary key of the identifying entity set, namely *course*, and *{sec_id, year, semester}* distinguishes *section* entities for the same course.

Figure E-R diagram with a weak entity set.

❖ In Figure, the weak entity set *section* depends on the strong entity set *course* via the relationship set *sec_course*.

In E-R diagrams, a weak entity set is depicted with a rectangle, like a strong entity set, but there are two main differences:

✓ The discriminator of a weak entity is underlined with a dashed, rather than a solid, line.

✓ Relationship set connecting weak entity set to identifying strong entity set is depicted by a double diamond.

The above figure also illustrates the use of double lines to indicate *total participation*; finally, the arrow from *sec_course* to *course* indicates that each section is related to a single course.

Note: A weak entity set can also participate in relationships other than the identifying relationship. A weak entity set may participate as owner in an identifying relationship with another weak entity set. It is also possible to have a weak entity set with more than one identifying entity set. A particular weak entity would then be identified by a combination of entities, one from each identifying entity set. The primary key of the weak entity set would consist of the union of the primary keys of the identifying entity sets, plus the discriminator of the weak entity set. A weak entity set representation more aptly models a situation where set participates in relationships than identifying relationship, and where weak entity set has several attributes.

REDUCTION TO RELATIONAL SCHEMAS

We can represent a database that conforms to an E-R database schema by a collection of relation schemas. For each entity set and for each relationship set in the database design, there is a unique relation schema to which we assign the name of the corresponding entity set or relationship set.

Both the E-R model and the relational database model are abstract, logical representations of real-world enterprises. Because the two models employ similar design principles, we can convert an E-R design into a relational design.

1. Representation of Strong Entity Sets with Simple Attributes: Let E be a strong entity set with only simple descriptive attributes $a_1, a_2, \ldots, a_n$. We represent this entity by a schema called E with n distinct attributes. Each tuple in a relation on this schema corresponds to one entity of the entity set E.

For schemas derived from strong entity sets, the primary key of the entity set serves as the primary key of the resulting schema. This follows directly from the fact that each tuple corresponds to a specific entity in the entity set.

As an example, consider the entity set *student* of above E-R diagram. This entity set has three attributes: *ID, name, tot_cred*. We represent this entity set by a schema called *student* with three attributes: *student* (*ID, name, tot_cred*)

Note that since student ID is primary key of entity set, it is also the primary key of the relation schema.

2. Representation of Strong Entity Sets with Complex Attributes: Composite attributes are handled by creating a separate attribute for each of the component attributes; we do not create a separate attribute for the composite attribute itself.

To illustrate, consider the *instructor* entity set. For the composite attribute *name*, the schema generated for *instructor* contains the attributes *first_name*, *middle_name*, and *last_name*; there is no separate attribute for *name*. Similarly, for the composite attribute *address*, the schema generated contains the attributes *street*, *city*, *state*, and *zip code*. Since *street* is a composite attribute it is replaced by *street number*, *street name*, and *apt number*. The relational schema derived from the version of entity set *instructor* with complex attributes, without including the multivalued attribute, is thus:

instructor (*ID, first_name, middle_name, last_name, street_number, street_name, apt_number, city, state, zip_code, date_of_birth*)

- ***Multivalued attributes are treated differently from other attributes.*** We have seen that attributes in an E-R diagram generally map directly into attributes for appropriate relation schemas. Multivalued attributes, however, are an exception; new relation schemas are created for these attributes. For a multivalued attribute *M*, we create a relation schema *R* with an attribute *A* that corresponds to *M* and attributes corresponding to primary key of the entity set or relationship set of which *M* is an attribute.

As an example, consider entity set *instructor*, which includes multivalued attribute *phone_number*. Primary key of *instructor* is *ID*. For this multivalued attribute, we create a relation schema *instructor_phone* (*ID, phone_number*). Each phone_number of an instructor is represented as a unique tuple in relation on this schema. Thus, if we had an instructor with *ID* 22222, and phone_numbers 9849012345 and 9848012345, the relation *instructor phone* would have two tuples (22222, 9849012345) and (22222, 9848012345).

We create a primary key of the relation schema consisting of all attributes of the schema. We also create a foreign-key on the relation schema created from multivalued attribute, with the primary key of the entity set referencing the relation generated from the entity set. In the above example, the foreign-key constraint on the *instructor_phone* relation would be that attribute *ID* references the *instructor* relation.

- Derived attributes are not explicitly represented in the relational data model. However, they can be represented as "methods" in other data models such as the object-relational data model

3. **Representation of Weak Entity Sets:** Let *A* be a weak entity set with attributes $a_1, a_2, \ldots, a_m$. Let *B* be the strong entity set with attributes $b_1, b_2, \ldots, b_n$ on which *A* depends. Let the primary key of *B* is b_1. We represent the entity set *A* by a relation schema called *A* as: $\{a_1, a_2, \ldots, a_m\} \cup \{b_1\}$

For schemas derived from a weak entity set, the combination of the primary key of the strong entity set and the discriminator of the weak entity set serves as the primary key of the schema. We also create a foreign-key constraint on the relation *A*, specifying that the attribute b_1 reference the primary key of the relation *B*. The foreign key constraint ensures that for each tuple representing a weak entity, there is a corresponding tuple representing the corresponding strong entity.

Ex: *section* (*course_id*, *sec_id*, *semester*, *year*)

✓ Because of the "on delete cascade" specification on the foreign key constraint, if a *course* entity is deleted, then so are all the associated *section* entities.

4. **Representation of Relationship Sets:** Let *R* be a relationship set, let $a_1, a_2, \ldots, a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in *R*, and let the descriptive attributes (if any) of *R* be $b_1, b_2, \ldots, b_n$. We represent this relationship set by a relation schema called *R* with one attribute for each member of the set: $\{a_1, a_2, \ldots, a_m\} \cup \{b_1, b_2, \ldots, b_n\}$

The primary key is chosen as follows:

- ❖ For a binary many-to-many relationship, the union of the primary-key attributes from the participating entity sets becomes the primary key.
- ❖ For a binary one-to-one relationship set, the primary key of either entity set can be chosen as the primary key.
- ❖ For a binary many-to-one or one-to-many relationship set, the primary key of the entity set on the "many" side of the relationship set serves as the primary key.
- ❖ For an *n*-ary relationship set without any arrows on its edges, the union of the primary key-attributes from the participating entity sets becomes the primary key.
- ❖ For an *n*-ary relationship set with an arrow on one of its edges, the primary keys of the entity sets not on the "arrow" side of the relationship set serve as the primary key for the schema. Recall that we allowed only one arrow out of a relationship set.

We can also create foreign-key constraints on the relation schema *R*. As an illustration, consider the relationship set *advisor* in the E-R diagram. This relationship set involves the following two entity sets:

- *instructor* with the primary key *ID*.
- *student* with the primary key *ID*.

Since the relationship set has no attributes, the *advisor* schema has two attributes, the primary keys of *instructor* and *student*. Since both attributes have the same name, we rename them *i_ID* and s*_ID*. Since the *advisor* relationship set is many-to- one from *student* to *instructor* the primary key for the *advisor* relation schema is *s ID*.

Note: The schema for the relationship set linking a weak entity set to its corresponding strong entity set is redundant and does not need to be present in a relational database design based upon an E-R diagram.

ENTITY-RELATIONSHIP DESIGN ISSUES

1. Use of Entity Sets versus Attributes: Two natural questions arise in a database design are: What constitutes an attribute, and what constitutes an entity set? Unfortunately, there are no simple answers.

The distinctions mainly depend on the structure of the real-world enterprise being modeled, and on the semantics associated with the attribute in question.

A common mistake is to use the primary key of an entity set as an attribute of another entity set, instead of using a relationship. Another related mistake that people sometimes make is to designate the primary-key attributes of the related entity sets as attributes of the relationship set. This should not be done since the primary-key attributes are already implicit in the relationship set.

2. Use of Entity Sets versus Relationship Sets: It is not always clear whether an object is best expressed by an entity set or a relationship set. One possible guideline in determining whether to use an entity set or a relationship set is to designate a relationship set to describe an action that occurs between entities. This approach can also be useful in deciding whether certain attributes may be more appropriately expressed as relationships.

3. Binary versus *n*-ary Relationship Sets: Relationships in databases are often binary. Some relationships that appear to be non-binary could be better represented by several binary relationships. It is always possible to replace a non-binary (n-ary, for $n > 2$) relationship set by a number of distinct binary relationship sets. For simplicity, consider the abstract ternary ($n = 3$) relationship set R, relating entity sets A, B, and C. We replace the relationship set R by an entity set E, and create three relationship sets as shown in figure.

RA, relating E and A.
RB, relating E and B.
RC, relating E and C.

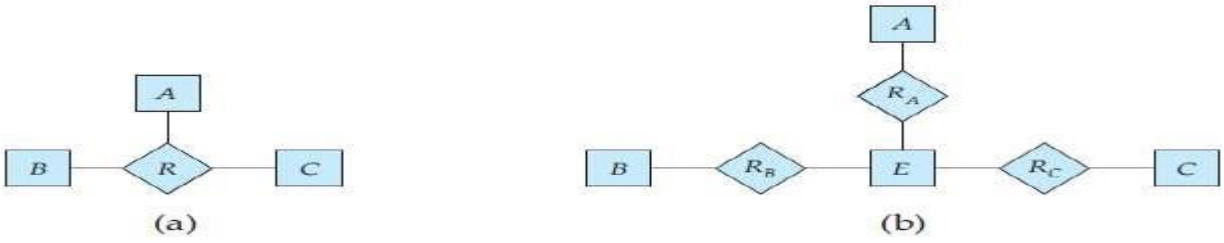


Figure Ternary relationship versus three binary relationships.

If the relationship set R had any attributes, these are assigned to entity set E; further, a special identifying attribute is created for E. For each relationship (a_i , b_i , c_i) in the relationship set R, we create a new entity e_i in the entity set E. Then, in each of the three new relationship sets, we insert a relationship as follows:

(e_i , a_i) in RA.
(e_i , b_i) in RB.
(e_i , c_i) in RC.

We can generalize this process in a straightforward manner to n-ary relationship sets. Conceptually, we can restrict E-R model to include only binary relationship sets. But this restriction is not always desirable.

- ✓ An identifying attribute may have to be created for the entity set created to represent the relationship set to increase the complexity of the design and overall storage requirements.
- ✓ An n-ary relationship set shows more clearly that several entities participate in a single relationship.
- ✓ There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships.

4. Placement of Relationship Attributes: The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set.

- ✓ For one-to-one relationship sets, relationship attribute can be associated with either of the participating entities.
- ✓ Attributes of a one-to-many relationship set can be repositioned to only the entity set on the "many" side of the relationship.
  - ■ The design decision of where to place descriptive attributes in such cases, as a relationship or entity attribute, should reflect the characteristics of the enterprise being modeled.
- ✓ The choice of attribute placement is more clear-cut for many-to-many relationship sets. When an attribute is determined by the combination of participating entity sets, rather than by either entity separately, that attribute must be associated with the many-to-many relationship set.

EXTENDED E-R FEATURES

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. The extended E-R features include: specialization, generalization, attribute inheritance, and aggregation.

- ❖ **Specialization:** "The process of designating sub groupings within an entity set is called **specialization**". An entity set may include sub groupings of entities that are distinct in some way from other entities in the set. The E-R model provides a means for representing these distinctive entity groupings. As an example, the entity set *person* may be further classified as one of the following:
  - *employee*.
  - *student*.

Each of these person types is described by a set of attributes that includes all attributes of entity set *person* plus possibly additional attributes. For example, *employee* entities may be described further by attribute *salary*, whereas *student* entities may be described further by attribute *tot_cred*. The specialization of *person* allows us to distinguish among person entities according to whether they correspond to employees or students: in general, a person could be an employee, a student, both, or neither.

We can apply specialization repeatedly to refine a design. For instance, university employees may be further classified as one of the following:

- *instructor*.
- *secretary*.

Each of these employee types is described by a set of attributes that includes all the attributes of entity set *employee* plus additional attributes. For example, *instructor* entities may be described further by the attribute *rank* while *secretary* entities are described by the attribute *hours per week*. An entity set may be specialized by more than one distinguishing feature. In our example, the distinguishing feature among employee entities is the job the employee performs.

We refer to this relationship as the ***ISA relationship***, which stands for "is a" and represents, for example, that an instructor "is a" employee. In terms of an E-R diagram, specialization is depicted by a hollow arrow-head pointing from the specialized entity to the other entity.

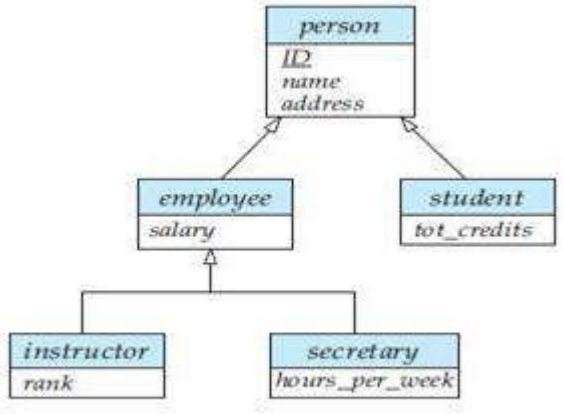


Figure Specialization and generalization.

The way we depict specialization in an E-R diagram depends on whether an entity may belong to multiple specialized entity sets or if it must belong to at most one specialized entity set. The former case (multiple sets permitted) is called **overlapping specialization**, while the latter case (at most one permitted) is called **disjoint specialization**. For an overlapping specialization (as is the case for *student* and *employee* as specializations of *person*), two separate arrows are used. For a disjoint specialization (as is the case for *instructor* and *secretary* as specializations of *employee*), a single arrow is used.

The specialization relationship may also be referred to as a **superclass-subclass** relationship. Higher- and lower-level entity sets also may be designated by the terms **superclass** and **subclass**, respectively. The refinement from an initial entity set into successive levels of entity sub groupings represents a **top-down** design process in which distinctions are made explicit.

- **Generalization:** "The design process in which multiple entity sets are synthesized into a higher-level entity set on the basis of common features is known as **generalization**". This is a **bottom-up** design process.

The database designer may have first identified:

- *instructor* entity set with attributes *instructor_id*, *instructor_name*, *instructor_salary*, and *rank.*
- *secretary* entity set with attributes *secretary_id*, *secretary_name*, *secretary_salary*, and *hours per week.*

There are similarities between the *instructor* entity set and the *secretary* entity set in the sense that they have several attributes that are conceptually the same across the two entity sets: namely, the id, name, and salary attributes. To create a generalization, the attributes must be given a common name and represented with the higher-level entity *employee*. In our example, *employee* is the higher-level entity set and *instructor* and *secretary* are lower-level entity sets. The *employee* entity set is the superclass of the *instructor* and *secretary* subclasses.

For all practical purposes, generalization is a simple inversion of specialization. We apply both processes, in combination, in the course of designing the E-R schema for an enterprise. In terms of the E-R diagram itself, we do not distinguish between specialization and generalization. New levels of entity representation are distinguished (specialization) or synthesized (generalization) as the design schema comes to express fully the database application and the user requirements of the database. Differences in the two approaches may be characterized by their starting point and overall goal.

Constraints on Generalizations: One type of constraint involves determining which entities can be members of a given lower-level entity set. Such membership may be one of the following:

1. **Condition-defined:** In condition-defined lower-level entity sets, membership is evaluated on the basis of whether or not an entity satisfies an explicit condition or predicate. Since all lower-level entities are evaluated on the basis of the same attribute this type of generalization is said to be **attribute-defined**.
2. **User-defined:** User-defined lower-level entity sets are not constrained by a membership condition; rather, the database user assigns entities to a given entity set. The assignment is implemented by an operation that adds an entity to an entity set.

❖ A second type of constraint relates to whether or not entities may belong to more than one lower-level entity set within a single generalization. The lower level entity sets may be one of the following:

1. **Disjoint:** A *disjoint constraint* requires an entity belong to no more than one lower-level entity set.
2. **Overlapping:** In *overlapping generalizations*, the same entity may belong to more than one lower-level entity set within a single generalization.

❖ A final constraint, the **completeness constraint** on a generalization or specialization, specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within the generalization/specialization. This constraint may be one of the following:

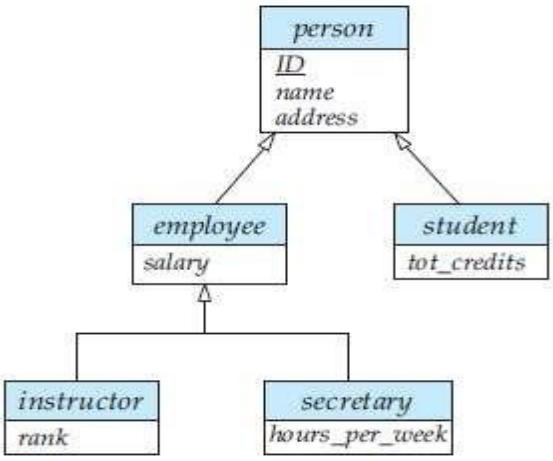
1. **Total Generalization** or **Specialization:** Each higher-level entity must belong to a lower-level entity set.
2. **Partial Generalization** or **Specialization:** Some higher-level entities may not belong to any lower-level entity set.

Note: Partial generalization is the default. We can specify total generalization in an E-R diagram by adding the keyword "total" in the diagram and drawing a dashed line from the keyword to the corresponding hollow arrow-head to which it applies (for a total generalization), or to the set of hollow arrow-heads to which it applies (for an overlapping generalization).

Attribute Inheritance: A crucial property of the higher- and lower-level entities created by specialization and generalization is ***attribute inheritance***. The attributes of the higher-level entity sets are said to be inherited by the lower-level entity sets.

For example, *student* and *employee* inherit the attributes of *person*. Thus, *student* is described by its *ID*, *name*, and *address* attributes, and additionally a *tot_cred* attribute; *employee* is described by its *ID*, *name*, and *address* attributes, and additionally a *salary* attribute. Attribute inheritance applies through all tiers of lower-level entity sets; thus, *instructor* and *secretary*, which are subclasses of *employee*, inherit the attributes *ID*, *name*, and *address* from *person*, in addition to inheriting *salary* from *employee*.

A lower-level entity set (or subclass) also inherits participation in the relationship sets in which its higher-level entity (or superclass) participates. Like attribute inheritance, participation inheritance applies through all tiers of lower-level entity sets. For example, suppose the *person* entity set participates in a relationship *person_dept* with *department*. Then, the *student*, *employee*, *instructor* and *secretary* entity sets, which are subclasses of the *person* entity set, also implicitly participate in the *person_dept* relationship with *department*.



The above figure depicts a **hierarchy** of entity sets. In the figure, *employee* is a lower-level entity set of *person* and a higher-level entity set of the *instructor* and *secretary* entity sets. In a hierarchy, a given entity set may be involved as a lower level entity set in only one ISA relationship; that is, entity sets in this diagram have only **single inheritance**. If an entity set is a lower-level entity set in more than one ISA relationship, then the entity set has **multiple inheritance**, and the resulting structure is said to be a ***lattice***.

- **Aggregation:** One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship proj_guide, between an instructor, student and project. Now suppose that each instructor guiding a student on a project is required to file a monthly evaluation report. We model the evaluation report as an entity evaluation, with a primary key evaluation id. One alternative for recording the (student, project, instructor) combination to which an evaluation corresponds is to create a quaternary (4-way) relationship set eval_for between instructor, student, project, and evaluation.

Using the basic E-R modeling constructs, we obtain the following E-R diagram.

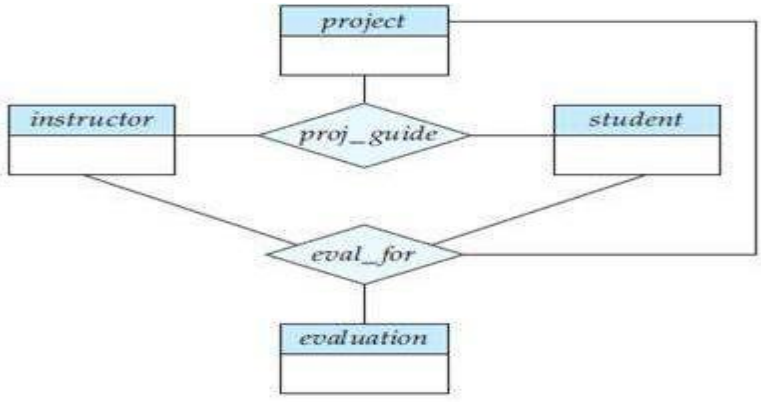


Figure E-R diagram with redundant relationships.

It appears that the relationship sets proj_guide and eval_for can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some instructor, student, project combinations may not have an associated evaluation.

There is ***redundant information*** in the resultant figure, however, since every instructor, student, project combination in eval_for must also be in proj_guide. The best way to model a situation such as the one just

described is to use aggregation. **Aggregation** is an abstraction through which relationships are treated as higher-level entities. Thus, for our example, we regard the relationship set proj_guide (relating the entity sets instructor, student, and project) as a higher-level entity set called proj_guide. Such an entity set is treated in the same manner as is any other entity set.

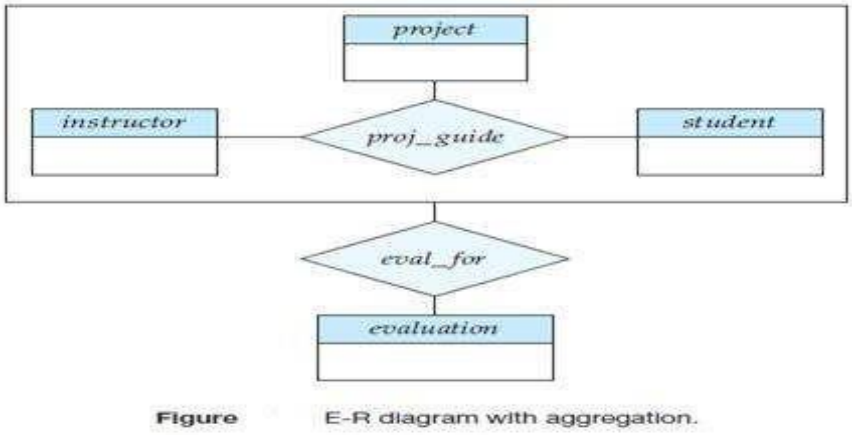


Figure E-R diagram with aggregation.

We can then create a binary relationship eval_for between proj_*guide* and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for, as shown in the figure.

REDUCTION TO RELATION SCHEMAS

Representation of Generalization: There are two different methods of designing relation schemas for an E-R diagram that includes generalization. We refer to Generalization figure and simplify it by including only the first tier of lower-level entity sets—that is, *employee* and *student*. We assume that *ID* is the primary key of *person*.

1) Create a schema for the higher-level entity set. For each lower-level entity set, create a schema that includes an attribute for each of the attributes of that entity set plus one for each attribute of the primary key of the higher-level entity set. Thus, for the E-R diagram of figure we have three schemas: *person* (*ID, name, street, city*), *employee* (*ID, salary*), *student* (*ID, tot_cred*)

❖ The primary-key attributes of the higher-level entity set become primary key attributes of all lower-level entity sets also which can be underlined. In addition, we create foreign-key constraints on the lower-level entity sets. In the above example, the *ID* attribute of *employee* would reference the primary key of *person*, and similarly for *student*.

2) An alternative representation is possible, if the generalization is disjoint and complete. Here, we do not create a schema for the higher-level entity set. Instead, for each lower level entity set, we create a schema that includes an attribute for each of the attributes of that entity set plus one for *each* attribute of the higher-level entity set. Then, for the E-R diagram of figure, we have two schemas: *employee* (*ID, name, street, city, salary*), *student* (*ID, name, street, city, tot_cred*)

Both these schemas have *ID*, which is the primary-key attribute of the higher level entity set *person*, as their primary key.

One drawback of second method lies in defining foreign-key constraints. To avoid this problem, we need to create a relation schema containing at least the primary-key attributes of same entity. If the second method were used for an overlapping generalization, some values would be stored multiple times.

Representation of Aggregation: Designing schemas for an E-R diagram containing aggregation is straightforward. Consider the diagram of Aggregation. Schema for the relationship set *eval_for* between the aggregation of *proj_guide* and the entity set *evaluation* includes an attribute for each attribute in the primary keys of the entity set *evaluation*, and the relationship set *proj_guide*. It also includes an attribute for any descriptive attributes, if they exist, of the relationship set *eval_for*. We then transform relationship sets and entity sets within the aggregated entity set following the rules we have already defined.

The rules for creating primary-key and foreign-key constraints on relationship sets can be applied to relationship sets involving aggregations as well, with the aggregation treated like any other entity set. The primary key of the aggregation is the primary key of its defining relationship set. No separate relation is required to represent the aggregation; the relation created from the defining relationship is used instead.

OTHER ASPECTS OF DATABASE DESIGN

Data Constraints and Relational Database Design: Constraints serve several purposes. The most obvious one is the automation of consistency preservation. By expressing constraints in the SQL DDL, the designer is able to ensure that the database system itself enforces the constraints. This is more reliable than relying on each application program individually to enforce constraints. It also provides a central location for the update of constraints and the addition of new ones.

A further advantage of stating constraints explicitly is that certain constraints are particularly useful in designing relational database schemas. Constraint enforcement comes at a potentially high price in performance each time the database is updated.

Data constraints are also useful in determining the physical structure of data. It may be useful to store data that are closely related to each other in physical proximity on disk so as to gain efficiencies in disk access. Certain index structures work better when the index is on a primary key.

Usage Requirements: Queries, Performance: Database system performance is a critical aspect of most enterprise information systems. There are two main metrics for performance:

1. ***Response time***—the amount of time a *single* transaction takes from start to finish in either the average case or the worst case.
2. ***Throughput***—the number of queries or updates (often referred to as *transactions*) that can be processed on average per unit of time.

Systems that process large numbers of transactions in a batch style focus on having high throughput. Systems that interact with people or time-critical systems often focus on response time. Most commercial database systems historically have focused on throughput; however, a variety of applications including Web-based applications require good response time.

Queries that involve joins require more resources to evaluate than those that do not. In cases where a join is required, the database administrator may choose to create an index that facilitates evaluation of that join. For queries—whether a join is involved or not—indices can be created to speed evaluation of selection predicates (SQL where clause) that are likely to appear.

Another aspect of queries that affects choice of indices is relative mix of update and read operations. While an index may speed queries, it also slows updates.

Authorization Requirements: Authorization constraints affect design of the database as well because SQL allows access to be granted to users on the basis of components of the logical design of the database. A relation schema may need to be decomposed into two or more schemas to facilitate the granting of access rights in SQL.

Data Flow, Workflow: The term *workflow* refers to the combination of data and tasks involved in processes like a CAD system. Workflows interact with the database system as they move among users and users perform their tasks on the workflow.

In addition to the data on which workflows operate, the database may store data about the workflow itself, including the tasks making up a workflow and how they are to be routed among users.

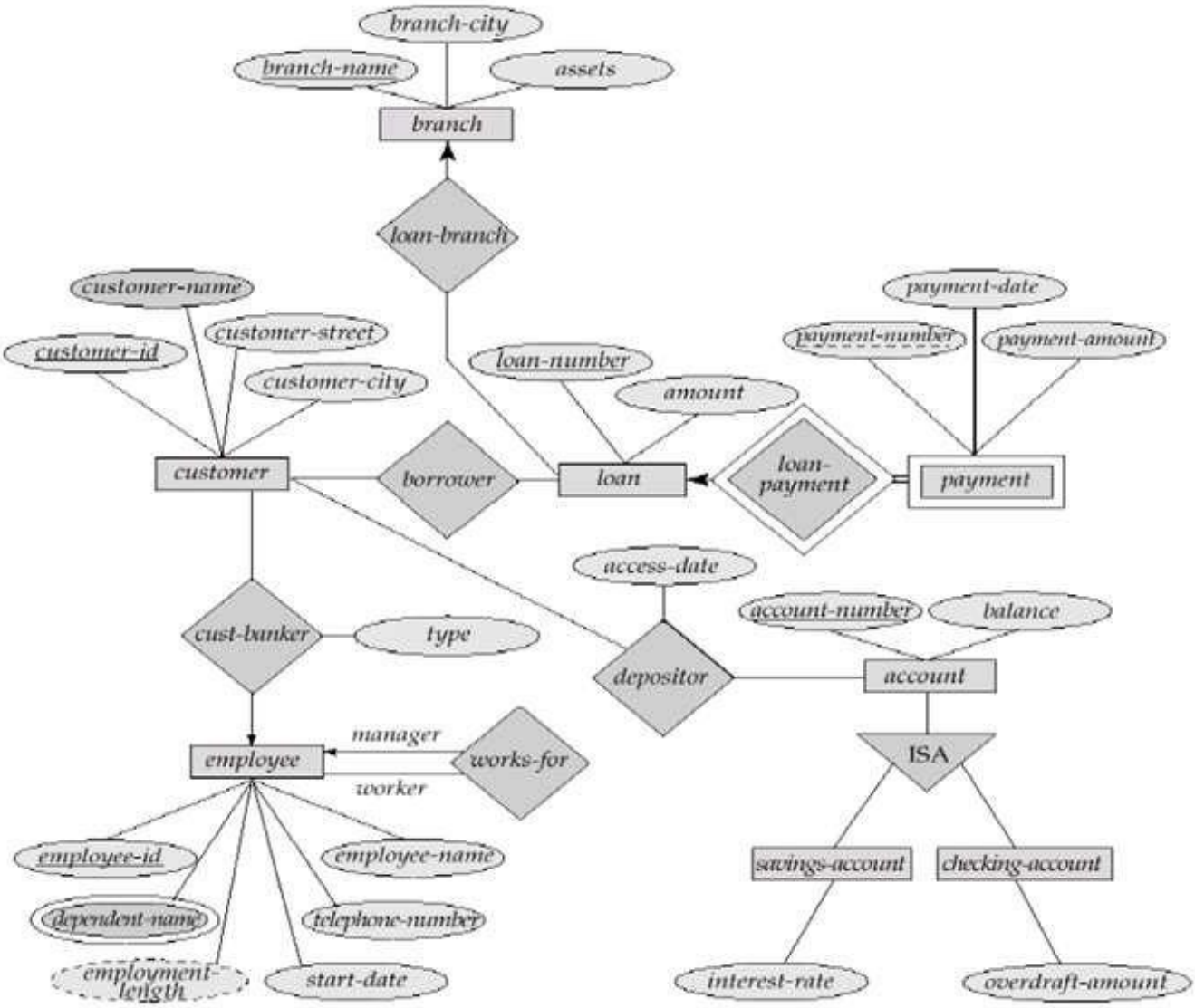
Other Issues in Database Design: Database design is usually not a one-time activity. The needs of an organization evolve continually, and the data that it needs to store also evolve correspondingly. During the initial database-design phases, the database designer may realize that changes are required at the conceptual, logical, or physical schema levels. Changes in the schema can affect all aspects of the database application. A good database design anticipates future needs of an organization, and ensures that the schema requires minimal changes as the needs evolve.

A good design should account not only for current policies, but should also avoid or minimize changes due to changes that are anticipated, or have a reasonable chance of happening.

Furthermore, the enterprise that the database is serving likely interacts with other enterprises and, therefore, multiple databases may need to interact. Conversion of data between different schemas is an important problem in real-world applications. The XML data model is widely used for representing data when it is exchanged between different applications.

Finally, it is worth noting that database design is a human-oriented activity in two senses: the end users of the system are people (even if an application sits between the database and the end users); and the database designer needs to interact extensively with experts in the application domain to understand the data requirements of the application. All the people involved with the data should be taken into account for a database design and deployment to succeed within the enterprise.

DATABASE DESIGN FOR BANKING ENTERPRISE



UNIT II

SYLLABUS: Relational Model: Structure of Relational Databases, Fundamental Relational-Algebra Operations, Additional Relational – Algebra Operations, Extended Relational - Algebra Operations, Null Values, Modification of the Databases.

Structured Query Language: Data Definition, Basic Structure of SQL Queries, Set Operations, Aggregate Functions, Null Values, Nested Sub-queries, Complex Queries, Views, Modification of the Database, Joined Relations.

RELATIONAL MODEL

Structure of Relational Database: A relational database consists of a collection of *tables*. Each table is assigned a unique name. A row in a table represents a *relationship* among a set of values. For basic structure, consider the *deposit* table of the following figure:

Branch-Name	Account-Number	Customer-Name	Balance
A	101	Dileep	2500
B	215	Kapil	6200
C	305	Ravi	3400
D	217	Raghu	7400
E	786	Bharath	4000

Table: The *deposit* relation

Domain: A set of permitted values of each attribute of a table is called as domain. The above table has four attributes: *Branch Name, Account Number, Customer Name*, and *Balance*. For each attribute, there is a set of permitted values, called the Domain of that attribute.

For example, for the attribute *Branch Name,* the *domain* is the set of all branch names. Similarly for the *Balance*, the domain is the all balance values.

Relation: A relation is subset of a Cartesian product of a list of domains.

Ex: In the above relation Deposit, there are four attributes. D1 is the domain of branch Names, D2 is the set of all account numbers, D3 is the set of all customer names and D4 is the set of all balance values.

Every row in a deposit relation consists of 4-tuples (v1, v2, v3, v4) where v1 is the branch Name (i.e., v1 is in domain D1), v2 is an account number (i.e., v2 is in domain D2), v3 is the customer name (i.e., v3 is in domain D3) and v4 is the balance (i.e., v4 is in domain D4). In general, deposit will contain only a subset of all possible rows.

i.e., Deposit is a subset of D1 X D2 X D3 X D4

TUPLE: Row of a given flat file (table) is called a tuple of the relation.

Ex: In the above relation, there are five tuples (rows), i.e., the given relation cardinality is five.

Degree of a Relation: The number of the attributes in a given relation is called degree of the relation.

Ex: The degree of the given relation is 4.

KEYS IN A RELATIONAL DATABASE

KEY: A data item (attribute) is used to identify or locate a record is called a key (entity identifier).

VARIOUS KEYS IN A RELATIONAL DATABASE

1) Primary key 2) Candidate key 3) Alternate key 4) Secondary key 5) Super key 6) Foreign key

1) **PRIMARY KEY:** The primary key is defined as a data item (attribute) which uniquely identifies a record (one row) in a relation.

Roll-No	Name	Date-of-Birth	Second-Language	Division
95	Swetha	31st Dec	Hindi	First
96	Dhatri	26th Jan	Sanskrit	Second
97	Shivani	15th Aug	Telugu	First
98	Kavya	01st Nov	Hindi	Second
99	Jagruti	29th Feb	Sanskrit	First
100	Shravani	26th Jan	Telugu	First

The *student* Relation

In the above relation *student*, we can choose roll number attribute as primary key because for each row, there is a unique value of Roll number attribute i.e., same roll number is not repeated in another rows.

Therefore *Roll Number* is primary key for given relation.

2) **CANDIDATE KEY:** In a relation in which there is more than one attribute combination possessing the unique identification property, all the various combinations of attributes, which serve as a primary key are called the candidate keys of the given relation.

Note: Subset of Candidate keys is not a Primary Key.

Ex:-In the given relation student:

{Roll number} can identify each row uniquely. So it is one candidate key.

{Second Language, Date of Birth} can identify each row uniquely; it is one more candidate key.

{Date of Birth, Division} can identify each row uniquely so it is one more candidate key.

{Roll Number, Date of Birth} can identify each row uniquely. But subset of this key is {Roll Number} {Date of Birth}.

So Here {Roll Number] is a primary key. As per definition, subset of candidate keys not a primary key. So {Roll Number, Date of Birth} is not a candidate key.

3) **ALTERNATE KEY:** A candidate key that is not the primary key, called as alternate key.

Ex: In a given relation student

{Roll number} is unique for every student. Similarly all student names are unique (no students having the same name). Here we can choose either Roll number or name as a primary key.

In such a case we may arbitrarily choose one of the candidates. Say *Roll Number* as the primary key for the relation. A candidate key that is not the primary key, such *Name* in the relation student is called alternate key.

i.e., if Roll number is primary key, then Name is alternate key. If Name is primary key, then Roll Number is alternate key.

4) **SECONDARY KEY:** System may also use a key which does not identify a unique record or tuple but which identifies all those which have certain property. This is referred to as a Secondary Key.

Ex: In a given relation STUDENT the value of the attribute "second language" may be used as a secondary key. This key could be used to identify those entities (students), who belong to second language Hindi, Telugu or Sanskrit.

Second-Language	Roll-No
Hindi	95
	98
Sanskrit	96
	99
Telugu	97
	100

Secondary Keys

5) **SUPER KEY:** More than one attribute combined together for unique identification of the record is defined as a Super Key As shown in figure below, neither supplier no. (S#), nor product no. (P#) are enough to identify the each row. To get unique information for each row, we need combined attributes s#, p#. i.e. {s# + p#} is a Super key (or) concatenate key.

S#	P#	Qty
S1	P1	500
S1	P2	700
S1	P3	450
S2	P4	920
S3	P1	650
S3	P5	400

Super Keys

6) **FOREIGN KEY:** A foreign key is an attribute or group of fields in a database record that points to a key field or group of fields forming a key of another database record in a different table. Usually a foreign key in one table refers to the primary key of another table. This way references can be made to link. For Ex: For an Account relation Customer_ID is considered as the foreign key.

QUERY LANGUAGES

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

a) Procedural

b) Non-procedural.

In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result. Relational algebra is called as procedural query language. In a nonprocedural language, the user describes the information desired without giving a specific procedure for obtaining the information. Tuple relational calculus and domain relational calculus is belong to nonprocedural query language. The relational algebra is procedural while the relational calculus and the domain relational calculus are nonprocedural.

RELATIONAL ALGEBRA

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project, union, set difference, Cartesian product,* and *rename*. In addition to the fundamental operations, there are several other operations—namely, *set intersection, natural join*, and *assignment*. We shall define these operations in terms of the fundamental operations.

Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations.

Consider the Two relations STUDENT, QUARTERLY. The STUDENT relation is used to describe the complete personal information about student, his roll no, name, date of birth, 2nd language. Another relation QUARTERLY used to describe the students marks in 3 subjects with roll no's.

STUDENT

Roll-No	Name	Date-of-Birth	Second-Language
1	Sunny	01-07-70	Hindi
2	Rashni	15-08-72	Sanskrit
3	Anthra	29-01-71	Hindi
4	Nasreen	31-12-70	Telugu

QUARTERLY

Roll-No	Maths	Physics	Computers
1	72	85	90
2	65	74	68
3	97	94	96
4	87	93	72

The following are queries based on relational algebra to obtain required information from stored relational database.

1) **The SELECT Operation:** The Select operation selects tuples that satisfy a given predicate. A lower case Greek letter sigma (σ) is used to denote Select operation. Predicate appears as subscript to r.

The argument relation is given in parentheses. The General form of selection operation is:

$$\sigma_{\text{predicate}}(\text{relation})$$

All comparisons =, #, <, >, <=, >= were allowed in the select operation predicate. Furthermore, several predicates may be combined into a large predicate using the connectives *and* (∧) and *or* (V).

Ex: 1) List out the complete information about all students whose 2nd language is Hindi

$$\sigma_{\text{2nd-language = Hindi}}(\text{STUDENT})$$

Result of the above Query is:

Roll-No	Name	Date-of-Birth	Second-Language
1	Sunny	01-07-70	Hindi
3	Anthra	29-01-71	Hindi

Ex: 2) Display all students with Roll no with their marks who secured more than 90 in all the three subjects

$$\sigma_{((\text{Maths} > 90) \wedge (\text{Physics} > 90) \wedge (\text{Computers} > 90))}(\text{QUARTERLY})$$

Result of the above Query is:

Roll-No	Maths	Physics	Computers
3	97	94	96

2) The PROJECT Operation: The projection of a relation is defined as a projection of all its tuples over some set of attributes. i.e., it yields a "vertical subset" of the relation. The projection operation is used to either reduce the number of attributes in the resultant relation or to reorder attributes. Projection is denoted by Greek letter pi (Π). We list these attributes that we wish to appear in the result as a subscript to. The argument relation follows in parenthesis.

General form of projection operation is

$$\Pi_{\text{List-of-attributes (Predicate)}}(\text{relation})$$

Ex: 1) List out all Roll Nos. and their Computer Marks

$$\Pi_{\text{Roll No, Computers}}(\text{QUARTERLY})$$

Result of the above Query is:

Roll-No	Computers
1	90
2	68
3	96
4	72

2) Display all the student names with their date of birth whose 2nd language is Hindi.

$$\Pi_{\text{Name, Date-of-birth}}(\sigma_{\text{2nd-language = Hindi}})(\text{STUDENT})$$

Result of the above query is:

Name	Date-of-Birth
Sunny	01-07-70
Anthra	29-01-71

3) What is the Date of Birth of Rashni?

$$\Pi_{\text{Date-of-Birth}} (\sigma_{\text{Name = Rashni}}) (\text{STUDENT})$$

Result of the above query is:

Date-of-Birth
15-08-72

4) Find all Roll Nos who secured more than 90 marks in Computers

$$\Pi_{\text{Roll-No}} (\sigma_{\text{computers} > 90}) (\text{QUARTERLY})$$

Result of the above query is:

Roll-No
1
3

3) The RENAME Operation: Unlike relations in the DB, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the **rename** operator, denoted by the lowercase Greek letter rho (ρ), lets us do this. Given a relational-algebra expression *E*, the following expression returns the result of expression *E* under the name *x*.

$$\rho_x (E)$$

Ex: $\rho_{\text{teacher}} (instructor)$

A relation *r* by itself is considered a (trivial) relational-algebra expression. Thus, we can also apply the rename operation to a relation *r* to get the same relation under a new name.

A Second form of the rename operation is as follows: Assume that a relational algebra expression *E* has arity *n*. Then, the following expression returns the result of expression *E* under the name *x*, and with the attributes renamed to *A*1, *A*2, . . . , *An*.

$$\rho_{x(A1,A2,\ldots,An)} (E)$$

Ex: $\rho_{\text{teacher (id, name, sal)}} (instructor)$

4) CARTESIAN PRODUCT Operation: This operation allows us to combine information from several relations. Thus operation is denoted by a cross(X). Thus operation is a binary. Suppose r1 and r2 are two relations, Cartesian product of these two relations can be written are r1 X r2.

In other words, Cartesian product of two relations is the concatenation of tuples belonging to the two relations. A new resultant relation scheme is created consisting of all possible combinations of tuples.

If there are m tuples in relation r1, and n tuples in relation r2, then there is m x n ways of choosing a pair of tuples. One tuple from each relation is chosen, so there are n1 x n2 tuples in r.

Ex: (i) Find student names and their Computer marks.

To list out the Name, Computer Marks, we have to refer both the relations, STUDENT & QUARTERLY. Student name is an attribute from STUDENT relation and Computers is an attribute from QUARTERLY relation. Referring 2 relations is denoted by "X"

STUDENT X QUARTERLY

Roll-No	Name	Date-of-Birth	Second-Language	Roll-No	Maths	Physics	Computers
1	Sunny	01-07-70	Hindi	1	72	85	90
1	Sunny	01-07-70	Hindi	2	65	74	68
1	Sunny	01-07-70	Hindi	3	97	94	96
1	Sunny	01-07-70	Hindi	4	87	93	72
2	Rashni	15-08-72	Sanskrit	1	72	85	90
2	Rashni	15-08-72	Sanskrit	2	65	74	68
2	Rashni	15-08-72	Sanskrit	3	97	94	96
2	Rashni	15-08-72	Sanskrit	4	87	93	72
3	Anthra	29-01-71	Hindi	1	72	85	90
3	Anthra	29-01-71	Hindi	2	65	74	68
3	Anthra	29-01-71	Hindi	3	97	94	96
3	Anthra	29-01-71	Hindi	4	87	93	72
4	Nasreen	31-12-70	Telugu	1	72	85	90
4	Nasreen	31-12-70	Telugu	2	65	74	68
4	Nasreen	31-12-70	Telugu	3	97	94	96
4	Nasreen	31-12-70	Telugu	4	87	93	72

Information is retrieved from the above relation STUDENT X QUARTERLY. By selecting the common attribute in the same relation i.e., in given two relations, Roll No is the common attribute.

STUDENT.ROLLNO = QUARTERLY.ROLLNO

$\Pi_{\text{Name, Computers}} (\sigma_{\text{STUDENT.ROLL-NO = QUARTERLY.ROLL-NO}})$ (STUDENT X QUARTERLY)

Name	Computers
Sunny	90
Rashni	68
Anthra	96
Nasreen	72

(ii) Find the Student Roll No, Date of Birth, 2nd Language, Maths, Physics and Computer Marks.

$\sigma_{\text{STUDENT.ROLL NO = QUARTERLY.ROLL NO.}}$ (STUDENT X QUARTERLY)

Result of the query is:

Roll-No	Name	Date-of-Birth	Second-Language	Roll-No	Maths	Physics	Computers
1	Sunny	01-07-70	Hindi	1	72	85	90
2	Rashni	15-08-72	Sanskrit	2	65	74	68
3	Anthra	29-01-71	Hindi	3	97	94	96
4	Nasreen	31-12-70	Telugu	4	87	93	72

5) **UNION Operation:** The union of two relations r and s is denoted by r ∪ s. The output relation Z= r ∪ s has tuples drawn from r and s. The result relation Z contains tuples that are in either r or s or in both of them. The duplicate tuples are eliminated.

For a union operation $r \cup s$ to be valid, we require that two conditions hold:

- The relations *r* and *s* must be of the same arity. i.e., they must have the same number of attributes.
- The domains of the *i*th attribute of *r* and the *i*th attribute of *s* must be the same, for all *i*.

Note that *r* and *s* can be either database relations or temporary relations that are the result of relational algebra expressions.

As an example, consider the relations CULTURAL (name, class) and SPORTS (name, class). These two relations represent information about all cultural competition winners & sports winners separately.

CULTURAL

Name	Class
Kavya	MPC III
Lahari	MSC II
Bhanu	MCA II
Zeba	MPC III
Nisha	MBA II
Hima	MPC III

SPORTS

Name	Class
Deepta	MCA III
Lahari	MSC II
Hima	MPC III
Archana	MBA II
Sheela	MPC III

Ex: Find all the student Names of MPC III who won cultural competition or sports competition or both competitions.

$$\Pi_{\text{Name}}(\sigma_{\text{Class = 'MPC III'}})(\text{CULTURAL}) \cup \Pi_{\text{Name}}(\sigma_{\text{Class = MPC III}})(\text{SPORTS})$$

NAME
Kavya
Zeba
Hima

NAME
Hima
Sheela

NAME
Kavya
Zeba
Hima
Sheela

6) **SET-DIFFERENCE Operation:** The difference between two relations r and s is r - s. The result relation contains the set of tuples belonging to r and not in s.

Ex: Find Student Names of MPC III who won cultural prizes but not sports.

$$\Pi_{\text{Name}}(\sigma_{\text{Class = 'MPC III'}})(\text{CULTURAL}) - \Pi_{\text{Name}}(\sigma_{\text{Class = MPC III}})(\text{SPORTS})$$

NAME
Kavya
Zeba
Hima

NAME
Hima
Sheela

NAME
Kavya
Zeba

Formal Definition of the Relational Algebra: The fundamental operations of relational algebra allow us to give a complete definition of an expression in the relational algebra. A basic expression in the relational algebra consists of either one of the following:

- A relation in the database
- A constant relation

A constant relation is written by listing its tuples within { }, for example
{(22222, Einstein, Physics, 95000), (76543, Singh, Finance, 80000)}.
A general expression in the relational algebra is constructed out of smaller sub expressions. Let $E1$ and $E2$ be relational-algebra expressions. Then, the following are all relational-algebra expressions:

- ✓ $E1 \cup E2$
- ✓ $E1 - E2$
- ✓ $E1 \times E2$
- ✓ $\sigma_P(E1)$, where P is a predicate on attributes in $E1$
- ✓ $\Pi_S(E1)$, where S is a list consisting of some of the attributes in $E1$
- ✓ $\rho_x(E1)$, where x is the new name for the result of $E1$

Additional Relational-Algebra Operations

We define additional operations that do not add any power to the algebra, but simplify common queries.

1) **SET-INTERSECTION Operation:** The intersection of two relations r and s is denoted by r ∩ s. The output relation contains the set of all tuples belonging to both r and s.

Ex: List out all the names belonging to MPC III who won both the cultural & sports competition.

$$\Pi_{\text{Name}}(\sigma_{\text{Class = 'MPC III'}})\,(\text{CULTURAL}) \cap \Pi_{\text{Name}}(\sigma_{\text{Class = MPC III}})\,(\text{SPORTS})$$

NAME
Kavya
Zeba
Hima

NAME
Hima
Sheela

NAME
Hima

2) **NATURAL JOIN Operation:** Natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the "join" symbol. Natural join operation forms a Cartesian product of its two arguments, performs a selection forcing equalities on those attributes that appear in both relation schemas, and finally removes duplicate columns.

Ex: STUDENT ⋈ QUARTERLY

This operation performs a Cartesian product (X) of two relations, performs selection equality on those attributes that appear in both the relation schemas and finally removes duplicate columns. In STUDENT X QUARTERLY relations ROLLNO is common in both relations.

i.e. STUDENT ⋈ QUARTERLY becomes:

Roll-No	Name	Date-of-Birth	Second-Language	Roll-No	Maths	Physics	Computers
1	Sunny	01-07-70	Hindi	1	72	85	90
2	Rashni	15-08-72	Sanskrit	2	65	74	68
3	Anthra	29-01-71	Hindi	3	97	94	96
4	Nasreen	31-12-70	Telugu	4	87	93	72

Query can be written as

$$\Pi_{\text{Roll No, Name, Date of Birth, 2nd language, Maths, Physics, Computers}} (\text{STUDENT} \bowtie \text{QUARTERLY})$$

Now, to find student names and Computer marks, the query will be:

$$\Pi_{\text{Name, Computers}} (\text{STUDENT} \bowtie \text{QUARTERLY})$$

Name	Computers
Sunny	90
Rashni	68
Anthra	96
Nasreen	72

3) **The ASSIGNMENT Operation:** It is convenient at times to write a relational-algebra expression by assigning parts of it to temporary relation variables. The **assignment** operation, denoted by ←, works like assignment in a programming language. To illustrate this operation, consider the definition of the natural-join operation. We could write $r \cup s$ as:

$$result \leftarrow r \cup s$$

The evaluation of an assignment does not result in any relation being displayed to the user. Rather, the result of the expression to the right of the ← is assigned to the relation variable on the left of the ←. This relation variable may be used in subsequent expressions.

For relational algebra queries, assignment must always be made to a temporary relation variable. Note that the assignment operation does not provide any additional power to the algebra. It is, however, a convenient way to express complex queries.

Extended Relational-Algebra Operations

The relational algebra operations that provide the ability to write queries that cannot be expressed using the basic relational-algebra operations are called **extended relational-algebra** operations.

1) **GENERALIZED PROJECTION:** The first operation is the **generalized-projection** operation, which extends the projection operation by allowing operations such as arithmetic and string functions to be used in the projection list. The generalized-projection operation has the form:

$$\Pi_{F1,F2,\ldots,Fn} (E)$$

where E is any relational-algebra expression, and each of $F1, F2, \ldots, Fn$ is an arithmetic expression involving constants and attributes in the schema of E. As a base case, the expression may be simply an attribute or a constant. In general, an expression can use arithmetic operations such as $+, -, *$, and $\div$ on numeric valued attributes, numeric constants, and on expressions that generate a numeric result. Generalized projection also permits operations on other data types, such as concatenation of strings.
For example, the expression:

$$\Pi_{ID,\ name,\ deptname,\ salary \div 12}(instructor)$$

gives the *ID, name, deptname,* and the monthly salary of each instructor.

2) **AGGREGATION:** The second extended relational-algebra operation is the aggregate operation *G,* which permits the use of aggregate functions such as min or average, on sets of values.

Aggregate Functions: Aggregate Functions take a collection of values and return a single value as a result. Aggregate operation in relational algebra is expressed as:

$$G_1, G_2, \ldots, Gn\ G_{F1(A1),\ F2(A2),\ldots,\ Fn(An)}(E)$$

Where, E is any relational algebra expression

Each F_i is an aggregate function

Each A_i is an attribute name

$G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)

The Aggregate Functions include:

1. **sum:** It is used to find the sum of values of an attribute in a relation.
   Ex: $G_{\mathbf{sum}(salary)}(instructor)$
2. **avg**: It is used to find the average value of an attribute in a relation.
   Ex: $G_{\mathbf{avg}(salary)}(instructor)$
3. **count**: It is used to find the number of values in an attribute in a relation.
   Ex: $G_{\mathbf{count}(salary)}(instructor)$

There are cases where we must eliminate multiple occurrences of a value before computing an aggregate function. If we do want to eliminate duplicates, we use the same function names as before, with the addition of the key word "**distinct**" appended to the end of the function name (for ex. **count-distinct**). Now, the above example can also be written as:

$G_{\mathbf{count\text{-}distnict}(salary)}(instructor)$

4. **min:** It is used to find the minimum value in an attribute in a relation.
   Ex: $G_{\mathbf{min}(salary)}(instructor)$
5. **max:** It is used to find the maximum value in an attribute in a relation.
   Ex: $G_{\mathbf{max}(salary)}(instructor)$

❖ Result of aggregation does not have a name.

- Can use rename operation to give it a name.
- For convenience, we permit renaming as part of aggregate operation

NULL VALUES: It is possible for tuples to have a null value, denoted by *null,* for some of their attributes. *null* signifies an unknown value or that a value does not exist.

- ✓ The result of any arithmetic expression involving *null* is *null.*
- ✓ Aggregate functions simply ignore null values
  - Is an arbitrary decision. Could have returned null as result instead.
  - We follow the semantics of SQL in its handling of null values.
- ✓ For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same. As an alternative, assume each null is different from each other like in SQL.

Arithmetic Operations and Comparisons with null values return the special truth value *unknown or null.* For logical operators with an input as null:

OR: (*unknown* **or** *true*) = *true*,
(*unknown* **or** *false*) = *unknown*
(*unknown* **or** *unknown) = unknown*

AND: *(true* **and** *unknown) = unknown,*
(false **and** *unknown) = false,*
(unknown **and** *unknown) = unknown*

NOT*: (***not** *unknown) = unknown*

MODIFYING THE DATABASE: The content of the database may be modified using the following operations:

- ❖ Deletion
- ❖ Insertion
- ❖ Updating

- All these operations are expressed using the assignment operator.

❖ **Deletion**: A delete request is expressed in much the same way as a query. However, instead of displaying tuples to the user, we remove the selected tuples from the database. We may delete only whole tuples; we cannot delete values on only particular attributes. In relational algebra, a deletion is expressed by:

$$r \leftarrow r - E$$

where r is a relation and E is a relational algebra query.

Ex: (i) Delete the information of the students whose second language is Hindi

$$STUDENT \leftarrow STUDENT - (\sigma_{second\ -language\ =\ Hindi})\ (STUDENT)$$

(ii) Delete the marks of the students who got less than 75 marks in COMPUTERS.

$$QUARTERLY \leftarrow QUARTERLY - (\sigma_{COMPUTERS < 75})\ (QUARTERLY)$$

- **Insertion**: To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. In relational algebra, an insertion is expressed by :

$$r \leftarrow r \cup E$$

where r is a relation and E is a relational algebra expression.

Ex: To insert the information of Mathew with Roll. No. 5, Date of Birth 15-9-1971 and Second Language is Telugu, we write:

$$STUDENT \leftarrow STUDENT \cup \{(5, \text{“15-9-71”}, \text{“Telugu”})\}$$

Updating: In certain situations we may wish to change a value in a tuple without changing all values in the tuple. It can be done by using the generalized projection operator.

$$r \leftarrow \Pi_{F1, F2, \ldots, Fn}(r)$$

Each *Fi* is either the *i*th attribute of *r*, if the *i*th attribute is not updated, or, which gives the new value for the attribute.

Ex: To increase the salary of each instructor by 10%, we write:

$$instructor \leftarrow \Pi_{sal * 1.10}(instructor)$$

SQL VERSUS RELATIONAL ALGEBRA

The term *select* in relational algebra has a different meaning than the one used in SQL. In relational algebra, the term *select* corresponds to what we refer to in SQL as *where*. We emphasize the different interpretations here to minimize potential confusion.

UNIT III
ADVANCED SQL

Advanced SQL: SQL Data Types and Schemas, Integrity Constraints, Authorization, Embedded SQL, Dynamic SQL, Functions and Procedural Constructs, Recursive Queries, Advanced SQL Features.

Relational Database Design: Features of Good Relational Design, Atomic Domains and First Normal Form, Functional-Dependency Theory, Decomposition using Functional Dependencies.

SQL Data Types and Schemas

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

- Can extract values of individual fields from date/time/timestamp
  Example: **extract** (**year from** r.starttime)

- Can cast string types to date/time/timestamp
  Example: **cast** <string-valued-expression> **as date**
  Example: **cast** <string-valued-expression> **as time**

User-Defined Types

create type construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2) final**

Usage:

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

create domain construct in SQL-92 creates user-defined domain types

create domain *person_name* **char**(20) **not null**

Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Domain Constraints

- Domain constraints are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  Example: create domain *Dollars* numeric (12, 2)
  create domain *Pounds* numeric (12,2)
- We cannot assign or compare a value of type Dollars to a value of type pounds.
  - However, we can convert type as below
    (cast *r.A* as *Pounds*)

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
    - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
    - **clob**: character large object -- object is a large collection of character data
    - When a query returns a large object, a pointer is returned rather than the large object itself.

Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes
    - Ex: **select ***
      from *student*
      where *ID* = '12345'

can be executed by using the index to find the required record, without looking at all records of *student*

Create Table Extensions

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

create table *temp instructor* **like** *instructor*;

The above statement creates a new table *temp instructor* that has the same schema as *instructor*

For example the following statement creates a table *t1* containing the results of a query.

create table *t1* **as**
(**select** *
from *instructor*
where *dept_name*= 'Music');

By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name.

Schemas, Catalogs, and Environments

Early file systems were flat; that is, all files were stored in a single directory. Current file systems, of course, have a directory (or, synonymously, folder) structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, /users/avi/db-book/chapter3.tex.

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**.

In order to perform any actions on a database, a user (or a program) must first *connect* to the database. The user must provide the user name and usually, a password for verifying the identity of the user. Each user has

a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for the connection; this corresponds to the current directory being set to the user's home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name may be used, for example,
catalog5.univ schema.course

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus if *catalog5* is the default catalog, we can use *univ schema.course* to identify the same relation uniquely. If a user wishes to access a relation that exists in a different schema than the default schema for that user, the name of the schema must be specified. However, if a relation is in the default schema for a particular user, then even the schema name may be omitted. Thus we can use just *course* if the default catalog is *catalog5* and the default schema is *univ schema*.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application—one a production version, other test versions—can run on the same database system. The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier (also referred to as the *authorization identifier*). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema.

We can create and drop schemas by means of **create schema** and **drop schema** statements. In most database systems, schemas are also created automatically when user accounts are created, with the schema name set to the user account name. The schema is created in either a default catalog, or a catalog specified in creating the user account. The newly created schema becomes the default schema for the user account. Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

Integrity Constraints

Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (*P*), where *P* is a predicate

Not Null Constraint

Ex: -- Declare *name* and *budget* to be **not null**
name **varchar**(20) **not null**
budget **numeric**(12,2) **not null**

-- Declare the domain *Dollars* to be **not null**

create domain *Dollars* **numeric**(12,2) **not null**

The Unique Constraint

unique (*A*1, *A*2, …, *A*m)
The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
Note: Candidate keys are permitted to be null (in contrast to primary keys).

primary key : both unique and not null
dept_name vachar(20) primary key;
or
primary key (dept_name);

The check clause

check (*P*), where *P* is a predicate
Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
   course_id varchar (8),
   sec_id varchar (8),
   semester varchar (6),
   year numeric (4,0),
   building varchar (15),
   room_number varchar (7),
   time slot id varchar (4),
   primary key (course_id, sec_id, semester, year),
   check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))
);
```

- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.
  - **create domain** *hourly_wage* **numeric(5,2)**
    constraint *value_test* **check**(*value* > = 4.00)
  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Default Values

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

```
create table student
(ID varchar (5),
name varchar (20) not null,
dept_name varchar (20),
tot cred numeric (3,0) default 0,
primary key (ID));
```

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

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

Referential Integrity

It Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
-- Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:

- The primary key clause lists attributes that comprise the primary key.
- The unique key clause lists attributes that comprise a candidate key.
- The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Example : *dept_name* **varchar**(20) **references** *department*
Or
create table *classroom* (*building* **varchar** (15), *room number* **varchar** (7), *capacity* **numeric** (4,0),
primary key (*building*, *room number*))

create table *department*
(*dept_name* **varchar** (20), *building* **varchar** (15), *budget* **numeric** (12,2) **check** (*budget* > 0),
primary key (*dept_name*))

create table *course*
(*course id* **varchar**(8),*title* **varchar**(50),*dept_name* **varchar**(20), *credits* **numeric** (2,0) **check** (*credits* > 0),
primary key (*course id*),
foreign key (*dept_name*) **references** *department*)

create table *instructor*
(*ID* **varchar** (5), *name* **varchar** (20), **not null** *dept_name* **varchar** (20),
salary **numeric** (8,2), **check** (*salary* > 29000),
primary key (*ID*),
foreign key (*dept_name*) **references** *department*)

create table *section*
(*course id* **varchar** (8), *sec id* **varchar** (8),
semester **varchar** (6), **check** (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'),
year **numeric** (4,0), **check** (*year* > 1759 and *year* < 2100)
building **varchar** (15), *room number* **varchar** (7), *time slot id* **varchar** (4),
primary key (*course id*, *sec id*, *semester*, *year*),
foreign key (*course id*) **references** *course*,
foreign key (*building*, *room number*) **references** *classroom*)

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back). However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *course*:

Cascading Actions in Referential Integrity

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

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, system updates the field *dept_name* in the referencing tuples in *course* to the new value as well.
SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

Integrity Constraint Violation During Transactions

Ex: **create table** *person* (
ID **char**(10),
name **char**(40),
mother **char**(10),
father **char**(10),
primary key *ID,*
foreign key *father* **references** *person,*
foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (next slide)

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone

Assertions: An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.

- An assertion in SQL takes the form
  - **create assertion** <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting
  for all *X*, *P*(*X*)
  is achieved in a round-about fashion using
  not exists *X* such that not *P*(*X*)

--Ex: update credits if passed in course.

create assertion *credits earned constraint* **check**
(**not exists** (**select** ID
from *student*
where *tot cred* <> (**select sum**(*credits*)
from *takes* **natural join** *course*
where *student.ID= takes.ID*
and *grade* **is not null and** *grade*<> 'F')

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

The **grant** statement is used to confer authorization

grant <privilege list> **on** <relation name or view name> **to** <user list>

<user list> is:

- a user-id
- **public**, which allows all valid users the privilege granted
- A role

Granting a privilege on a view does not imply granting any privileges on the underlying relations. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: grant users *U*1, *U*2, and *U*3 **select** authorization on the *branch* relation:
    grant select on department **to** *U*1*, U*2*, U*3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

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

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

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

Revoking Authorization in SQL: The **revoke** statement is used to revoke authorization.

revoke <privilege list> **on** <relation name or view name> **from** <user list>
Example: **revoke select on** *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;

- All privileges that depend on the privilege being revoked are also revoked.
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

Roles

- **create role** instructor;
- **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

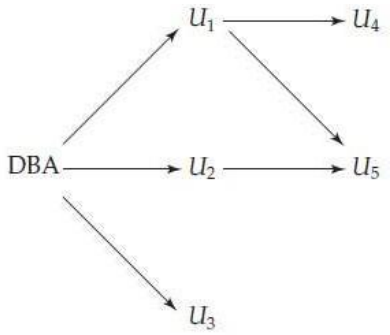
- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;

- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;

The passing of a specific authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users. Consider the graph for update authorization on *teaches*. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on *teaches* to U_j . The root of the graph is the database administrator.



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

Revoking of Privileges

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

- **revoke select on** *department* **from** Amit, Satoshi **cascade**;
- **revoke select on** *department* **from** Amit, Satoshi **restrict**;

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

- **revoke grant option for select on** *department* **from** Amit;

Cascading revocation is inappropriate in many situations. Suppose Satoshi has the role of *dean*, grants *instructor* to Amit, and later the role *dean* is revoked from Satoshi (perhaps because Satoshi leaves the university); Amit continues to be employed on the faculty, and should retain the *instructor* role.

To deal with the above situation, SQL permits a privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default, the current role associated with a session is null (except in some special cases). The current role associated with a session can be set by executing **set role** *role name*. The specified role must have been granted to the user; else the **set role** statement fails. To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

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

Accessing SQL from a Programming Language

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

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

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

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

1. **Dynamic SQL:** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.
2. **Embedded SQL:** Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor. The preprocessor submits SQL statements to the database system for pre compilation and optimization; then it replaces the SQL statements in the application program with appropriate code and function calls before invoking the programming-language compiler.

Embedded SQL

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

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
SQL is designed so that queries written in it can be optimized automatically and executed efficiently—and providing the full power of a programming language makes automatic optimization exceedingly difficult.

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

The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol. A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL. The basic form of these languages follows that of the System R embedding of SQL into PL/I.

EXEC SQL statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement > END_EXEC

Note: this varies by language (for example, the Java embedding uses

SQL { <embedded SQL statement > };)

Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

Before executing any sql statement, the program must first connect to the database as:

EXEC SQL **connect to** server_name **user** user_name END-EXEC

To write a relational query, we use the **declare cursor** statement. The result of the query is not yet computed. Rather, the program must use the **open** and **fetch** commands to obtain the result tuples.

Example: From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount.

- Specify the query in SQL and declare a *cursor* for it

EXEC SQL
declare *c* **cursor for**
select *ID, name*

from *student*
where tot_cred > *:credit_amount*
END_EXEC

The **open** statement causes the query to be evaluated
EXEC SQL **open** *c* END_EXEC
The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
EXEC SQL **fetch** *c* **into** :si*, :sn* END_EXEC

Repeated calls to **fetch** get successive tuples in the query result
A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

The **close** statement causes the database system to delete the temporary relation that holds the result of the query.
EXEC SQL **close** *c* END_EXEC

These details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

SQLJ, the Java embedding of SQL, provides a variation, where Java iterators are used in place of cursors. SQLJ associates the results of a query with an iterator, and the next() method of the Java iterator interface can be used to step through the result tuples, just as the preceding examples use **fetch** on the cursor. Embedded SQL expressions for database modification (**update**, **insert**, and **delete**) do not return a result. Thus, they are somewhat simpler to express. A database modification request takes the form

EXEC SQL < any valid **update, insert,** or **delete**> END-EXEC

Embedded SQL allows a host-language program to access the database, but it provides no assistance in presenting results to the user or in generating reports.

Dynamic SQL

Allows programs to construct and submit SQL queries at run time.
-- Example of the use of dynamic SQL from within a C program.

```
char * sqlprog = "update instrucotor
                  set sal = sal * 1.05
                where ID = ?"
EXEC SQL prepare dynprog from :sqlprog;
char ins [10] = "201101";
EXEC SQL execute dynprog using :instructor;
```

The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

ODBC and JDBC

API (application-program interface) for a program to interact with a database server
Application makes calls to
- Connect with the database server
- Send SQL commands to the database server
- Fetch tuples of result one-by-one into program variables

ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
JDBC (Java Database Connectivity) works with Java

ODBC: Open DataBase Connectivity(ODBC) is a standard for application program to communicate with a database server. It is an Application Program Interface (API) to
- open a connection with a database,
- send queries and updates,
- get back results.

Applications such as GUI, spreadsheets, etc. can use ODBC

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.
- Good programming requires checking results of every function call for errors;

JDBC: JDBC is a Java API for communicating with database systems supporting SQL. It supports a variety of features for querying and updating data, and for retrieving query results

- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

Procedural Extensions and Stored Procedures

SQL provides a **module** language that permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

Stored Procedures

- Can store procedures in the database
- then execute them using the **call** statement
- permit external applications to operate on the database without knowing about internal details

Functions and Procedures: SQL:1999 supports functions and procedures. Functions/procedures can be written in SQL itself, or in an external programming language

--Functions are particularly useful with specialized data types such as images and geometric objects

- Example: functions to check if polygons overlap, or to compare images for similarity

-- Some database systems support **table-valued functions**, which can return a relation as a result

SQL:1999 also supports a rich set of imperative constructs, including Loops, if-then-else, assignment constructs. Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions: *Define a function that, given the name of a department, returns the count of the number of instructors in that department.*

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

- *Find the department name and budget of all departments with more than 12 instructors.*

```
        select dept_name, budget
    from department
    where dept_count (dept_name ) > 1
```

SQL Procedures

- *The dept_count function could instead be written as procedure:*

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

- *Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.*

```
        declare d_count integer;
    call dept_count_proc( 'Physics', d_count);
```

- *Procedures and functions can be invoked also from dynamic SQL*
- *SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ*

Procedural Constructs

1) *Compound statement: **begin … end**,*
   - *May contain multiple SQL statements between **begin** and **end**.*
   - *Local variables can be declared within a compound statements*

2) ***While** and **loop** statements:*

```
        declare n integer default 0;
        while n < 10 do
           set n = n + 1
        end loop
        loop
    set n = n − 1
        until n = 0
        end loop
```

3) ***For** loop*
   - *Permits iteration over all results of a query*
   - *Example:*

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

4) *Conditional statements (**if-then-else**)*

*SQL:1999 also supports a **case** statement similar to C case statement*

- *Example procedure: registers student after ensuring classroom capacity is not exceeded*
  - *Returns 0 on success and -1 if capacity is exceeded*
  - *See book for details*
- *Signaling of exception conditions, and declaring handlers for exceptions*

```
        declare out_of_classroom_seats condition
declare exit handler for out_of_classroom_seats
begin
…
   .. signal out_of_classroom_seats
end
```

- *The handler here is **exit** -- causes enclosing **begin..end** to be exited*

Recursion in SQL: *SQL:1999 permits recursive view definition*
Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )

select ∗ from rec_prereq;
```

This example view, rec_prereq, is called the transitive closure of the prereq relation

The Power of Recursion

- *Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.*

Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of prereq with itself

- *This can give only a fixed number of levels of prereq*
- *Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work*
- *Alternative: write a procedure to iterate as many times as required*

- *Computing transitive closure using iteration, adding successive tuples to rec_prereq*
  - *The next we show a prereq relation*
  - *Each step of the iterative process constructs an extended version of rec_prereq from its recursive definition.*
  - *The final result is called the fixed point of the recursive view definition.*
- *Recursive views are required to be **monotonic**. That is, if we add tuples to prereq the view rec_prereq contains all of the tuples it contained before, plus possibly more*

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

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

Recursive queries should not use any of the following constructs as they make the query nonmonotonic.

1) *Aggregation on the recursive view*
2) *Not exists on a subquery that uses the recursive view*
3) *Set difference (except) whose right hand side uses the recursive view.*

Advanced SQL Features

Create table extensions: *In SQL 1999 to Create a table with the same schema as an existing table: create table temp_ins like instructor*
it requires 2 steps 1) creation and 2) store data.

In SQL 2003 we have a provision that combines both the steps into one: Create table ti as < sql query> ***with data***

More on Sub Queries:

--SQL:2003 allows subqueries to occur anywhere a value is required provided the subquery returns only one value. This applies to updates as well
--SQL:2003 allows subqueries in the from clause to access attributes of other relations in the from clause using the lateral construct:

```
        select name, salary, avg_salary
        from instructor I1,
    lateral (select avg(salary) as avg_salary
             from instructor I2
             where I2.dept_name= I1.dept_name);
```

When sub queries are written in either select or where clause we can access the attributes of outer query but when the sub query is written in from clause we cannot access. So to facilitate this we use lateral clause.

RELATIONAL DATABASE DESIGN

Features of Good Relational Design

Schema for the university database.

1. *classroom*(*building*, *room number*, *capacity*)
2. *department*(*dept_name*, *building*, *budget*)
3. *course*(*course id*, *title*, *dept_name*, *credits*)
4. *instructor*(*ID*, *name*, *dept_name*, *salary*)
5. *section*(*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)
6. *teaches*(*ID*, *course id*, *sec id*, *semester*, *year*)
7. *student*(*ID*, *name*, *dept_name*, *tot cred*)
8. *takes*(*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
9. *advisor*(*s ID*, *i ID*)
10. *time slot*(*time slot id*, *day*, *start time*, *end time*)
11. *prereq*(*course id*, *prereq id*)

Design Alternatives: Larger-Schema

- Suppose we combine *instructor* and *department* into *inst_dept*
  - *(No connection to relationship set inst_dept)*

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

- Result is possible repetition of information

A Combined Schema without Repetition

- Consider combining relations
  - *sec_class(sec_id, building, room_number)* and
  - *section(course_id, sec_id, semester, year)*
- into one relation
  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

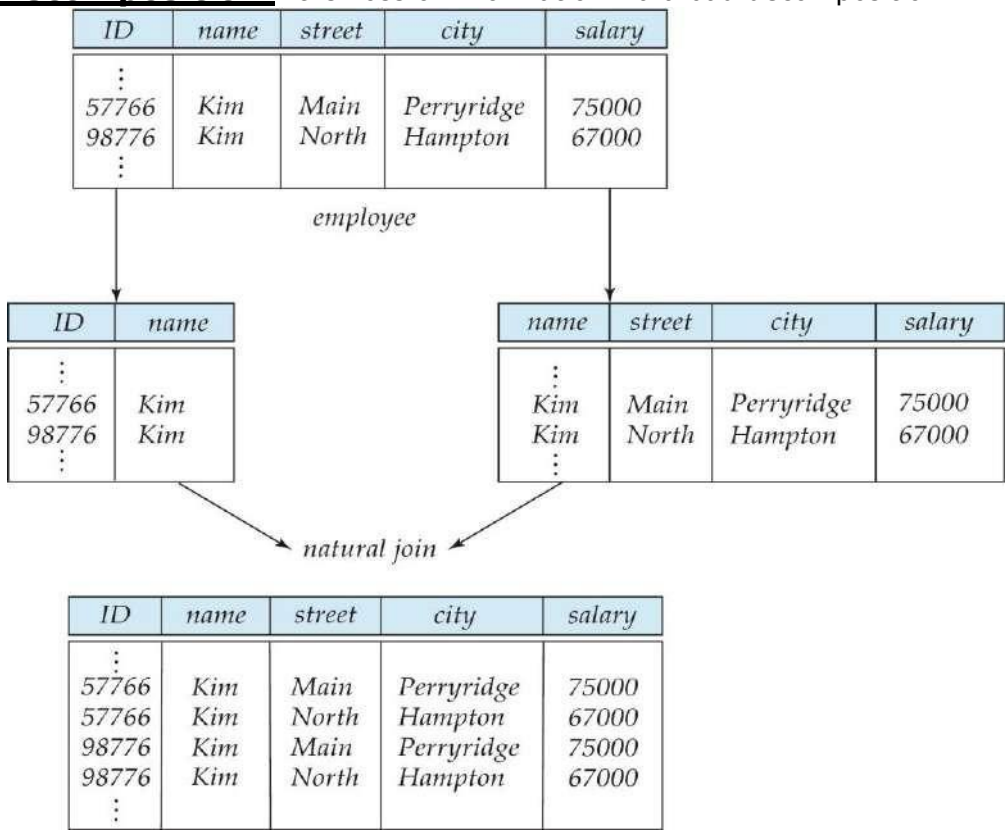
Design Alternatives: Smaller-Schema

- Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:
  - *dept_name* $\rightarrow$ *building, budget*
- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose *employee(ID, name, street, city, salary)* into

- *employee1* (*ID*, *name*)
- *employee2* (*name*, *street, city, salary*)

➢ The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition: It is Loss of information via a bad decomposition.



Our decomposition is unable to represent certain important facts of University employees. So we need to avoid this, and it is said to be **lossy decomposition** and on the other hand if we are able to get by original data then it is called as a **lossless decomposition**.

Example of **Lossless join decomposition**

- Decomposition of *R = (A, B, C) into* $R_1 = (A, B)$ $R_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

Functional Dependencies

➢ Constraints on the set of legal relations.
➢ Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
➢ A functional dependency is a generalization of the notion of a *key.*

Definition: Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. It is a relationship between attributes of a table dependent on each other. It helps in preventing data redundancy and gets to know about bad designs**.** Functional dependencies are constraints on the set of legal relations.
Syntax: A -> B
Ex: Account no -> Balance for account table.

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$, then the functional dependency $\alpha \rightarrow \beta$ holds on R if and only if for any legal relations r(R), whenever any two tuples t1 and t2 of r agree on the attributes of α, they also agree on the attributes of β. That is,

$$t1[\alpha] = t2\ [\alpha] \Rightarrow t1[\beta\ ] = t2\ [\beta\ ]$$

Example: Consider *r*(A,*B*) with the following instance of *r.*

A	B
1	4
1	5
3	7

On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

- *K* is a superkey for relation schema *R* if and only if $K \rightarrow R$
- *K* is a candidate key for *R* if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  inst_dept (ID, name, salary, dept_name, building, budget).

  We expect these functional dependencies to hold:

  dept_name → building and

  ID → building

  but would not expect the following to hold: dept_name → salary

Use of Functional Dependencies: We use functional dependencies to:

- test relations to see if they are legal under a given set of functional dependencies.
  -- If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
- specify constraints on the set of legal relations

  -- We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*

Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

For example, a specific instance of *instructor* may, by chance, satisfy *name* $\rightarrow$ *ID.*

Closure of a Set of Functional Dependencies

Given a set *F* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

The set of **all** functional dependencies logically implied by *F* is the ***closure*** of *F*.

- We denote the *closure* of *F* by F+.
- F+ is a superset of *F*.

Partial Dependency: If proper subset of candidate key determines non-prime attribute, it is called partial dependency.

Transitive Dependency: When an indirect relationship causes functional dependency it is called Transitive Dependency.

If P -> Q and Q -> R is true, then P-> R is a transitive dependency.

Trivial Dependency: If a functional dependency (FD) $X \rightarrow Y$ holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.

Non-Trivial Dependency: If an FD $X \rightarrow Y$ holds, where Y is not a subset of X, then it is called a non-trivial FD.

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. Normalization is used for mainly two purposes.

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored to maintain data consistency.

Problem without Normalization: Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not normalized.

- **Updation Anamoly:** We may want to update data in record, but it may exist in different tables or at different places due to redundancy.
- **Insertion Anamoly:** We may try to insert a new record but data may not be fully available or that record itself doesn't exist. Eg: to insert student details without knowing his department leads to Insertion Anamoly.
- **Deletion Anamoly:** When we try to delete a record it might have been saved or exists in some other place of database due to redundancy.

Normalization helps to remove anomalies and ensure that database is in consistent state.

Normalization Types:

Normalization types are divided into following normal forms.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First normal form (1NF): A relation is in first normal form if every attribute in every row can contain an atomic (only one single) value. An attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

Domain is atomic if its elements are considered to be indivisible units

Examples of non-atomic domains:

- Set of names, composite attributes
- Identification numbers like CS101 that can be broken up into parts

A relational schema R is in first normal form if the domains of all attributes of R are atomic. Non-atomic values complicate storage and encourage redundant storage of data.

- ***We assume all relations are in first normal form***

Second normal form (2NF): A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-prime attributes are fully functional dependent on the primary key

An attribute that is not part of any candidate key is known as non-prime attribute.

A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal Form(3NF): A relation schema *R* is in **third normal form** (3NF) if for all: $\alpha \rightarrow \beta$ in *F+* with at least one of the following holds:

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
2. α is a superkey for *R*
3. Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R.*

(**NOTE***:* each attribute may be in a different candidate key)

If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold). And, Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

Boyce-Codd Normal Form(BCNF):

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F+* of the form

$\alpha \rightarrow \beta$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
2. α is a superkey for *R*

Example schema *not* in BCNF: *instr_dept* (*ID, name, salary, dept_name, building, budget*)

because *dept_name*$\rightarrow$ *building, budget* holds on *instr_dept,* but *dept_name* is not a super key

Decomposing a Schema into BCNF

Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose *R* into:

1. (α U β)
2. (*R* - (β - α))

In our example,

α = *dept_name*

β = *building, budget*

and *inst_dept* is replaced by

- (α U β) = (*dept_name, building, budget*)
- (*R* - (β - α)) = (*ID, name, salary, dept_name*)

BCNF and Dependency Preservation

Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation. If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.* Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

How good is BCNF?

There are database schemas in BCNF that do not seem to be sufficiently normalized

Ex:1 Consider a relation

inst_info (ID, child_name, phone)

where an instructor may have more than one phone and can have multiple children

ID	*child_name*	*phone*
99999	David	512-555-1234
99999	David	512-555-4321
99999	William	512-555-1234
99999	Willian	512-555-4321

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

  (99999, David, 981-992-3443)

  (99999, William, 981-992-3443)

- Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999	David
99999	David
99999	William
99999	Willian

inst_phone

ID	*phone*
99999	512-555-1234
99999	512-555-4321
99999	512-555-1234
99999	512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)

Ex: Consider a database: *classes* (*course, teacher, book*) such that (*c, t, b*) $\in$ *classes* means that *t* is qualified to teach *c,*and *b* is a required textbook for *c*
The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

course	Teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	pete	Stallings

Classes

There are no non-trivial functional dependencies and therefore the relation is in BCNF
Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted
(database, Marilyn, DB Concepts)
(database, Marilyn, Ullman)

Therefore, it is better to decompose *classes* into:
Teaches:

Course	teacher
database	Avi
database	Hunk
database	Sudarshan
operating systems	Avi
operating systems	Jim

Text:

Course	Book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF),

Multivalued Dependencies (MVDs): Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \rightarrow\rightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

Example: Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets. *Y, Z, W*

- We say that $Y \rightarrow\rightarrow Z$ (Y **multidetermines** Z) if and only if for all possible relations r (R)
  $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$
  Then $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$
- Note that since the behavior of Z and W are identical it follows that
  $Y \rightarrow\rightarrow Z$ if $Y \rightarrow\rightarrow W$
- In our example:
  $ID \rightarrow\rightarrow child\_name$
  $ID \rightarrow\rightarrow phone\_number$
- The above formal definition is supposed to formalize the notion that given a particular value of Y (ID) it has associated with it a set of values of Z *(child_name)* and a set of values of W *(phone_number)*, and these two sets are in some sense independent of each other.
  Note: If $Y \rightarrow Z$ then $Y \rightarrow\rightarrow Z$

Fourth Normal Form (4NF): A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

- $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
- α is a superkey for schema R

- If a relation is in 4NF it is in BCNF

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Goals of Normalization

Let R be a relation scheme with a set F of functional dependencies.

- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, decompose it into a set of relation scheme {$R1, R2, ..., Rn$} such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

Functional-Dependency Theory

We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies. We then develop algorithms to generate lossless decompositions into BCNF and 3NF And we then develop algorithms to test if a decomposition is dependency-preserving.

Closure of a Set of Functional Dependencies: Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow$ C

The set of all functional dependencies logically implied by *F* is the ***closure*** of *F*.
We denote the *closure* of *F* by *F+.*

We can find all of F+ by applying **Armstrong's Axioms**:

1. if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
2. if $\alpha \rightarrow \beta$, then $\gamma\,\alpha \rightarrow \gamma\,\beta$ **(augmentation)**
3. if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

These rules are
- **sound** (generate only functional dependencies that actually hold) and
- **complete** (generate all functional dependencies that hold).

Example:

Let R = (A, B, C, G, H, I) and fd are given by
F = { $A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H$}
some members of *F+*
- $A \rightarrow H$
  - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
- $AG \rightarrow I$
  - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
- $CG \rightarrow HI$
  - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity.

We can further simplify manual computation of *F+* by using the following additional rules.
1. If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
2. If $\alpha \rightarrow \beta\,\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
3. If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\,\beta \rightarrow \delta$ holds, then $\alpha\,\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

The above rules can be inferred from Armstrong's axioms.

Algorithm for Computing F+ :

To compute the closure of a set of functional dependencies F:

```
F + = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F +
    for each pair of functional dependencies f1and f2 in F +
        if f1 and f2 can be combined using transitivity
            then add the resulting functional dependency to F +
until F + does not change any further
```

Closure of Attribute Sets: Given a set of attributes a, define the *closure* of a under *F* (denoted by a+) as the set of attributes that are functionally determined by a under *F*

Algorithm to compute a+, the closure of a under *F*

```
result := a;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ
        end
```

Example of Attribute Set Closure

Let R = (A, B, C, G, H, I) and fd's are given by

$F = \{A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

Now to compute $(AG)+$ we trace above algorithm to get :

1. *result = AG*
2. *result = ABCG* $(A \rightarrow C$ and $A \rightarrow B)$
3. *result = ABCGH* $(CG \rightarrow H$ and $CG \subseteq AGBC)$
4. *result = ABCGHI* $(CG \rightarrow I$ and $CG \subseteq AGBCH)$

Is *AG* a candidate key?

1. Is AG a super key?
    1. Does $AG \rightarrow R$? == Is $(AG)+ \supseteq R$
2. Is any subset of AG a superkey?
    1. Does $A \rightarrow R$? == Is $(A)+ \supseteq R$
    2. Does $G \rightarrow R$? == Is $(G)+ \supseteq R$

Uses of Attribute Closure:

There are several uses of the attribute closure algorithm:

1. Testing for super key:
    a. To test if α is a superkey, we compute $\alpha+$, and check if $\alpha+$ contains all attributes of R.
2. Testing functional dependencies
    a. To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in $F+$), just check if $\beta \subseteq \alpha+$.
    b. That is, we compute $\alpha+$ by using attribute closure, and then check if it contains β.
    c. Is a simple and cheap test, and very useful
3. Computing closure of F
    a. For each $\gamma \subseteq R$, we find the closure $\gamma+$, and for each $S \subseteq \gamma+$, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover:

Sets of functional dependencies may have redundant dependencies that can be inferred from the others.

For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C\}$

Parts of a functional dependency may be redundant

Ex:: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Ex:: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies.

Canonical Cover Definition: A *canonical cover* for *F* is a set of dependencies *Fc* such that

1. *F* logically implies all dependencies in *Fc,* and
2. *Fc* logically implies all dependencies in *F,* and
3. No functional dependency in *Fc* contains an extraneous attribute, and
4. Each left side of functional dependency in *Fc* is unique.

Extraneous Attributes:

Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.

1. Attribute A is **extraneous** in α if $A \in \alpha$
   and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
2. Attribute *A* is **extraneous** in β if $A \in \beta$
   and the set of functional dependencies
   $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F.*

Note: implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one

Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).

Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$

C is extraneous in $AB \rightarrow CD$ since AB $\rightarrow C$ can be inferred even after deleting C.

Testing if an Attribute is Extraneous: *Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.*

1) *To test if attribute $A \in \alpha$ is extraneous in α*
   a) *compute $(\{\alpha\} - A)+$ using the dependencies in F*
   b) *check that $(\{\alpha\} - A)+$ contains β; if it does, A is extraneous in α*
2) *To test if attribute $A \in \beta$ is extraneous in β*
   a) *compute $\alpha+$ using only the dependencies in*
      $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
   b) *check that $\alpha+$ contains A; if it does, A is extraneous in β*

Computing a Canonical Cover:

To compute a canonical cover for F:

repeat
  Use the union rule to replace any dependencies in F
    $\alpha 1 \rightarrow \beta 1$ and $\alpha 1 \rightarrow \beta 2$ with $\alpha 1 \rightarrow \beta 1\ \beta 2$
  Find a functional dependency $\alpha \rightarrow \beta$ with an
    extraneous attribute either in α or in β
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until F does not change

Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Let R = (A, B, C) and
$F = \{A \rightarrow BC$
$B \rightarrow C$
$A \rightarrow B$
$AB \rightarrow C\}$

Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

1. A is extraneous in $AB \rightarrow C$
   a. Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
      i. Yes: in fact, $B \rightarrow C$ is already present!
   b. Set is now $\{A \rightarrow BC, B \rightarrow C\}$
2. C is extraneous in $A \rightarrow BC$
   a. Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
      i. Yes: using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
         1. Can use attribute closure of A in more complex cases
3. The canonical cover is: $\{ A \rightarrow B , B \rightarrow C \}$

Decomposition using functional dependencies

Lossless-join Decomposition: For the case of $R = (R1, R2)$, we require that for all possible relations r on schema R

$$r = \prod R1\ (r) \bowtie \prod R2\ (r)$$

A decomposition of R into $R1$ and $R2$ is lossless join if and only if at least one of the following dependencies is in $F+$:

1. $R1 \cap R2 \rightarrow R1$
2. $R1 \cap R2 \rightarrow R2$

The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example: *Let R = (A, B, C) and F = {A → B, B → C)*
It Can be decomposed in two different ways

1. *R*1 = *(A, B), R*2 = *(B, C)*
   a. Lossless-join decomposition:
      i. *R*1 ∩ *R*2 = {*B*} and *B* → *BC*
   b. Dependency preserving
2. *R1 = (A, B), R*2 = *(A, C)*
   a. Lossless-join decomposition:
      i. *R*1 ∩ *R*2 = {*A*} and *A* → A*B*
   b. Not dependency preserving
      (cannot check *B* → *C* without computing *R1* *R*2)

Dependency Preservation: Let *Fi* be the set of dependencies *F* + that include only attributes in *Ri.* Decomposition is dependency preserving, if
(*F*1 ∪ *F*2 ∪ … ∪ *F*n)+ = *F* +

If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation: To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of *R* into *R*1, *R*2, …, *R*n we apply the following test (with attribute closure done with respect to *F*)

```
result = α
while (changes to result) do
    for each Ri in the decomposition
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
```

If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.

- We apply the test on all dependencies in *F* to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute *F*+ and (*F*1 ∪ *F*2 ∪ … ∪ *F*n)+

Example:
Let R = (*A, B, C*) and *F* = {*A* → *B* , *B* → *C*}
Key = {*A*}
R is not in BCNF
Decomposition *R*1 = (*A, B), R*2 = *(B, C)*

a. *R*1 and *R*2 in BCNF
b. Lossless-join decomposition
c. Dependency preserving

Testing for BCNF

To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF

1. Compute α+ (the attribute closure of α), and
2. Verify that it includes all attributes of *R*, that is, it is a super key of *R*.

Simplified test: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in *F*+.

- If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in *F*+ will cause a violation of BCNF either.

However, using only *F* is incorrect when testing a relation in a decomposition of R
Consider *R* = (*A, B, C, D, E*), with *F* = { *A* → *B, BC* → *D*}

- Decompose *R* into *R*1 = (*A,B*) and *R*2 = (*A,C,D, E*)
- Neither of the dependencies in *F* contain only attributes from (*A,C,D,E*) so we might be mislead into thinking *R*2 satisfies BCNF.
- In fact, dependency *AC* → *D* in *F*+ shows *R*2 is not in BCNF.

Testing Decomposition for BCNF: To check if a relation *Ri* in a decomposition of *R* is in BCNF,

a. Either test Ri for BCNF with respect to the restriction of F to Ri (that is, all FDs in F+ that contain only attributes from Ri)
b. or use the original set of dependencies *F* that hold on *R*, but with the following test:
    1. for every set of attributes $\alpha \subseteq Ri$, check that α+ (the attribute closure of α) either includes no attribute of *Ri*- α, or includes all attributes of *Ri*.
    i. If the condition is violated by some $\alpha \rightarrow \beta$ in *F*, the dependency
    $\alpha \rightarrow (\alpha+ - \alpha) \cap Ri$
    can be shown to hold on *Ri*, and *Ri* violates BCNF.
    ii. We use above dependency to decompose *Ri*

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving
Let R = (J, K, L) and $F = \{JK \rightarrow L, \quad L \rightarrow K\}$
Two candidate keys = *JK* and *JL*

1. *R* is not in BCNF
2. Any decomposition of *R* will fail to preserve
    1. $JK \rightarrow L$

This implies that testing for $JK \rightarrow L$ requires a join

Third Normal Form: There are some situations where

- BCNF is not dependency preserving, and
- efficient checking for FD violation on updates is important

Solution: define a weaker normal form, called Third Normal Form (3NF)

- Allows some redundancy (with resultant problems; we will see examples later)
- But functional dependencies can be checked on individual relations without computing a join.
- There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

Consider Relation R: let *R = (J, K, L)* and $F = \{JK \rightarrow L, L \rightarrow K\}$

- Two candidate keys: *JK* and *JL*
- *R* is in 3NF
    - $JK \rightarrow L$ *JK* is a superkey
      $L \rightarrow K$ *K* is contained in a candidate key

- Relation *dept_advisor*:
    - *dept_advisor* (*s_ID, i_ID, dept_name)*
      $F = \{s\_ID, dept\_name \rightarrow i\_ID, i\_ID \rightarrow dept\_name\}$
    - Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*
    - *R* is in 3NF
        - $s\_ID, dept\_name \rightarrow i\_ID$ *s_ID*
            - *dept_name* is a superkey
        - $i\_ID \rightarrow dept\_name$
            - *dept_name* is contained in a candidate key

Redundancy in 3NF: There is some redundancy in this schema

Example of problems due to redundancy in 3NF

Let R = (J, K, L) and $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
J1	L1	K1
J2	L1	K1
J3	L1	K1
null	L2	K2

- repetition of information (Ex:, the relationship *l*1, *k*1)
    - (*i_ID, dept_name)*

- need to use null values (Ex:, to represent the relationship
  *l*2, *k*2 where there is no corresponding value for *J*).
  - (*i_ID, dept_nameI)* if there is no separate relation mapping instructors to departments

Testing for 3NF: Optimization: Need to check only FDs in *F*, need not check all FDs in *F*+.

1. Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a super key.
2. If α is not a super key, we have to verify if each attribute in β is contained in a candidate key of *R*
   a. this test is rather more expensive, since it involve finding candidate keys
   b. testing for 3NF has been shown to be NP-hard
   c. Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

Comparison of BCNF and 3NF: It is always possible to decompose a relation into a set of relations that are in 3NF such that:

1. the decomposition is lossless
2. the dependencies are preserved

It is always possible to decompose a relation into a set of relations that are in BCNF such that:

- the decomposition is lossless
- it may not be possible to preserve dependencies.

Design Goals: Goal for a relational database design is:

1. BCNF.
2. Lossless join.
3. Dependency preservation.

If we cannot achieve this, we accept one of

- Lack of dependency preservation
- Redundancy due to use of 3NF

Interestingly, SQL does not provide a direct way of specifying functional dependencies other than super keys. Can specify FDs using assertions, but they are expensive to test that Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Overall Database Design Process

We have assumed schema *R* is given

- *R* could have been generated when converting E-R diagram to a set of tables.
- *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
- Normalization breaks *R* into smaller relations.
- *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

UNIT IV

INDEXING AND HASHING

Indexing and Hashing: Basic Concepts, Ordered Indices, B+ tree Index Files, B-tree Index Files, Multiple-Key Access, Static Hashing, Dynamic Hashing, Comparison of Ordered Indexing and Hashing, Bitmap Indices.

Index Definition in SQL Transactions: Transaction Concepts, Transaction State, Implementation of Atomicity and Durability, Concurrent Executions, Serializability, Recoverability, Implementation of Isolation, Testing for Serializability

Basic Concepts: Indexing mechanisms used to speed up access to desired data. EX: author catalog in library

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

Search key	pointer

Index files are typically much smaller than the original file. Two basic kinds of indices are:

1. **Ordered indices:** search keys are stored in sorted order
2. **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Index Evaluation Metrics:

- Access types supported efficiently. EX:
  - Ex: records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values (EX: 10000 < *salary* < 40000)
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

In an **ordered index,** index entries are stored sorted on the search key value. EX: author catalog in library.

Primary index: In a sequentially ordered file, the index whose search key specifies the sequential order of the file. It is also called **clustering index.** Here search key of a primary index is usually but not necessarily the primary key.

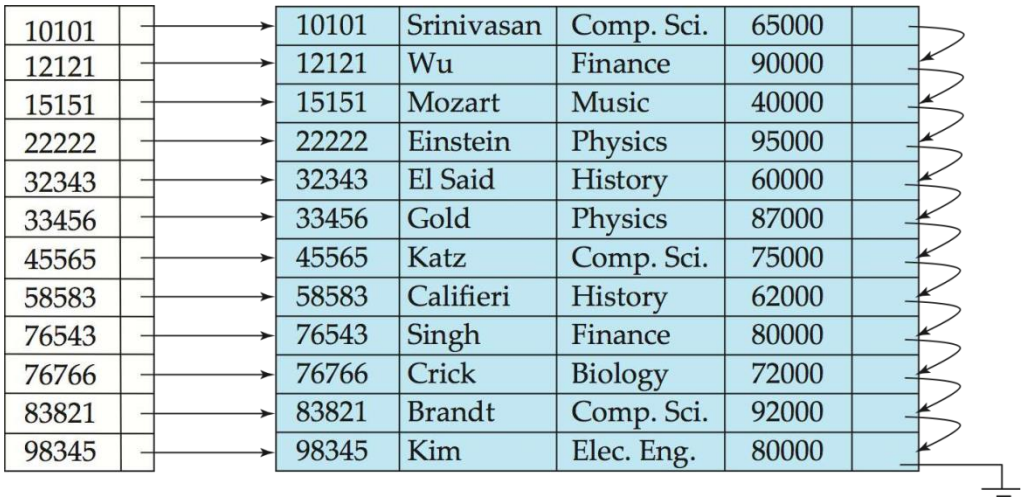
Secondary index: An index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index.**

Index-sequential file: ordered sequential file with a primary index.

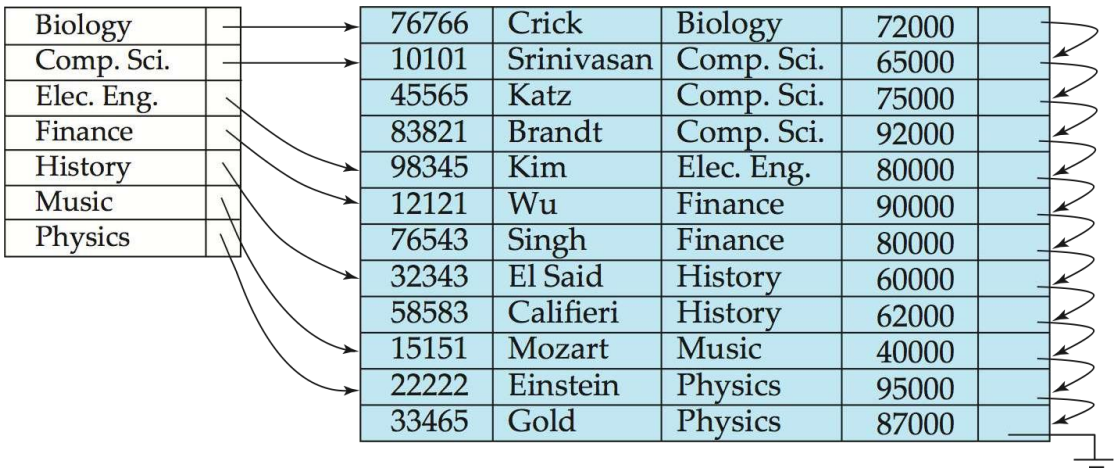
Dense Index Files

Dense index — Index record appears for every search-key value in the file.

EX: Index on *ID* attribute of *instructor* relation



Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



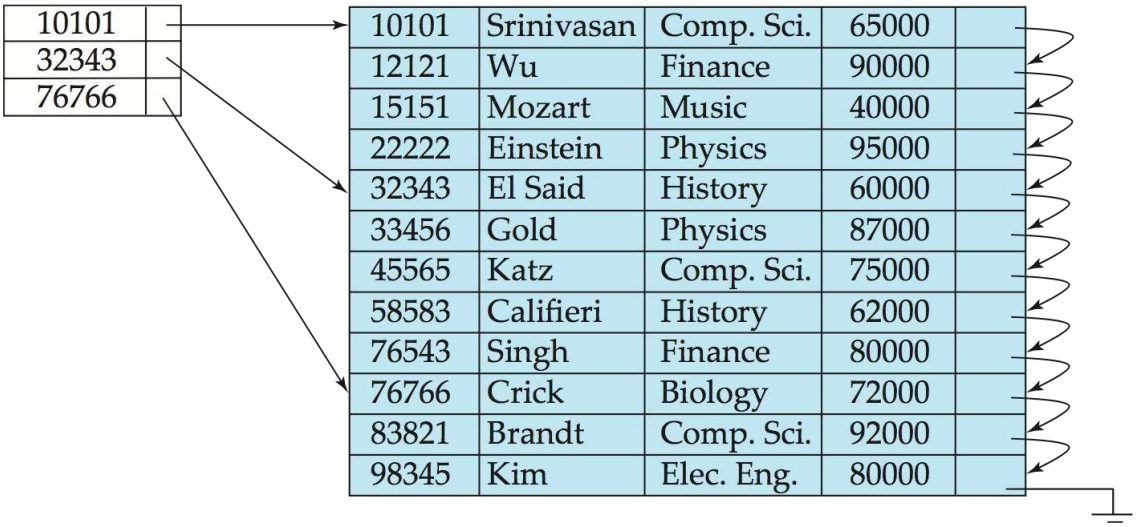
Sparse Index Files

Sparse Index: contains index records for only some search-key values.

It is Applicable when records are sequentially ordered on search-key

To locate a record with search-key value *K* we:

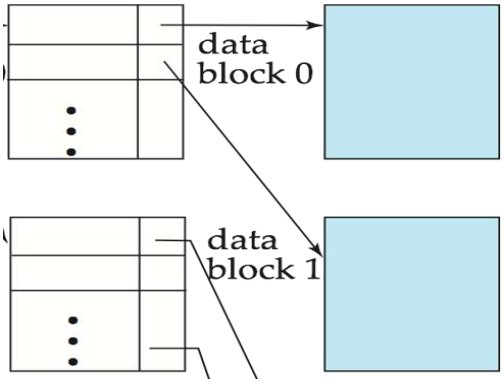
- Find index record with largest search-key value < *K*
- Search file sequentially starting at the record to which the index record points



Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.

Good tradeoff: Sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Secondary Indices Example

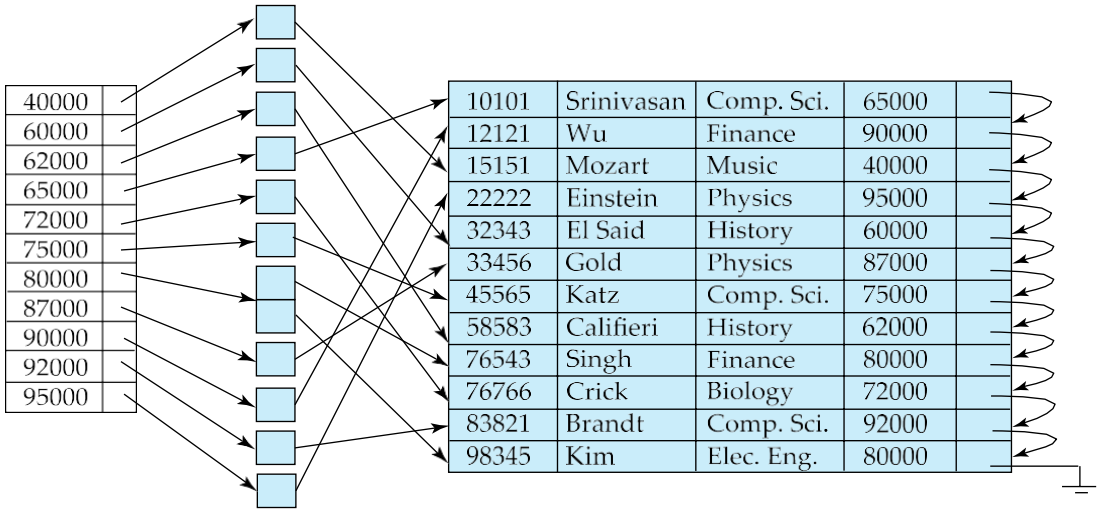


Fig: Secondary index on *salary* field of *instructor*

Index record points to a bucket that contains pointers to all the actual records with that particular search-key value. And Secondary indices have to be dense.

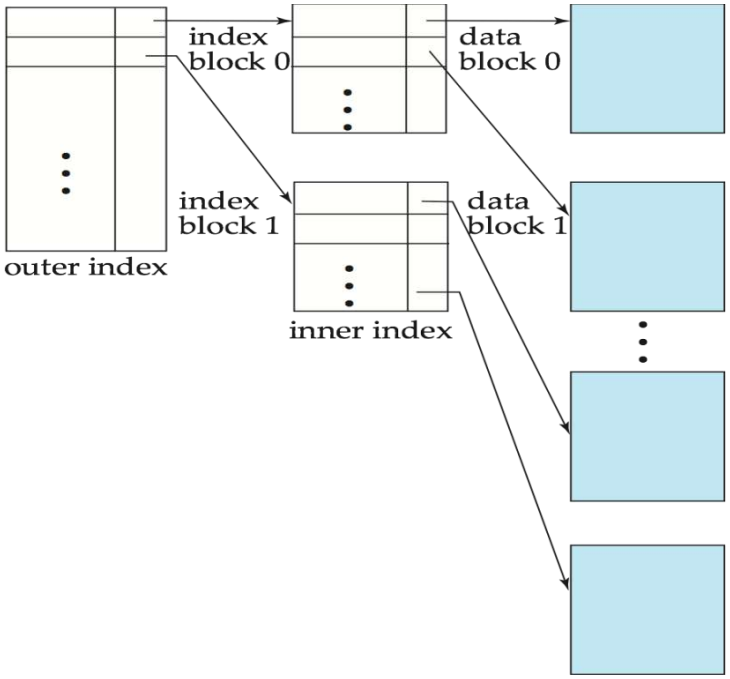
Primary and Secondary Indices offer substantial benefits when searching for records. But updating indices imposes overhead on database modification. When a file is modified, every index on the file must be updated.

- ✓ Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive.
- ✓ Each record access may fetch a new block from disk
- ✓ Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

Multilevel Index: If primary index does not fit in memory, access becomes expensive.
Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- outer index – a sparse index of primary index
- inner index – the primary index file

If even outer index is too large to fit in main memory, yet another level of index can be created, and so on. Indices at all levels must be updated on insertion or deletion from the file.



Index Update: Record Deletion

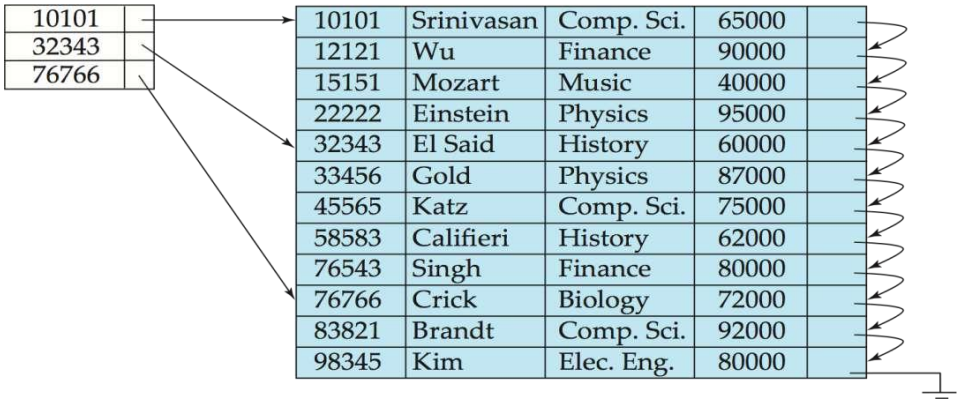
If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

Single-level index deletion:

Dense indices – Deletion of search-key: similar to file record deletion.

Sparse indices –

- If deleted key value exists in the index, the value is replaced by the next search-key value in the file (in search-key order).
- If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Record Insertion

Single-level index insertion: Perform a lookup using the key value from inserted record

Dense indices – if the search-key value does not appear in the index, insert it.

Sparse indices – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.

If a new block is created, the first search-key value appearing in the new block is inserted into the index.

- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

B+ Tree Index Files

B+ tree indices are an alternative to indexed-sequential files.

Disadvantage of indexed-sequential files:

- Performance degrades as file grows, since many overflow blocks get created.
- Periodic reorganization of entire file is required.

Advantage of B+ tree index files:

- Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- Reorganization of entire file is not required to maintain performance.

(Minor) Disadvantage of B+ trees:

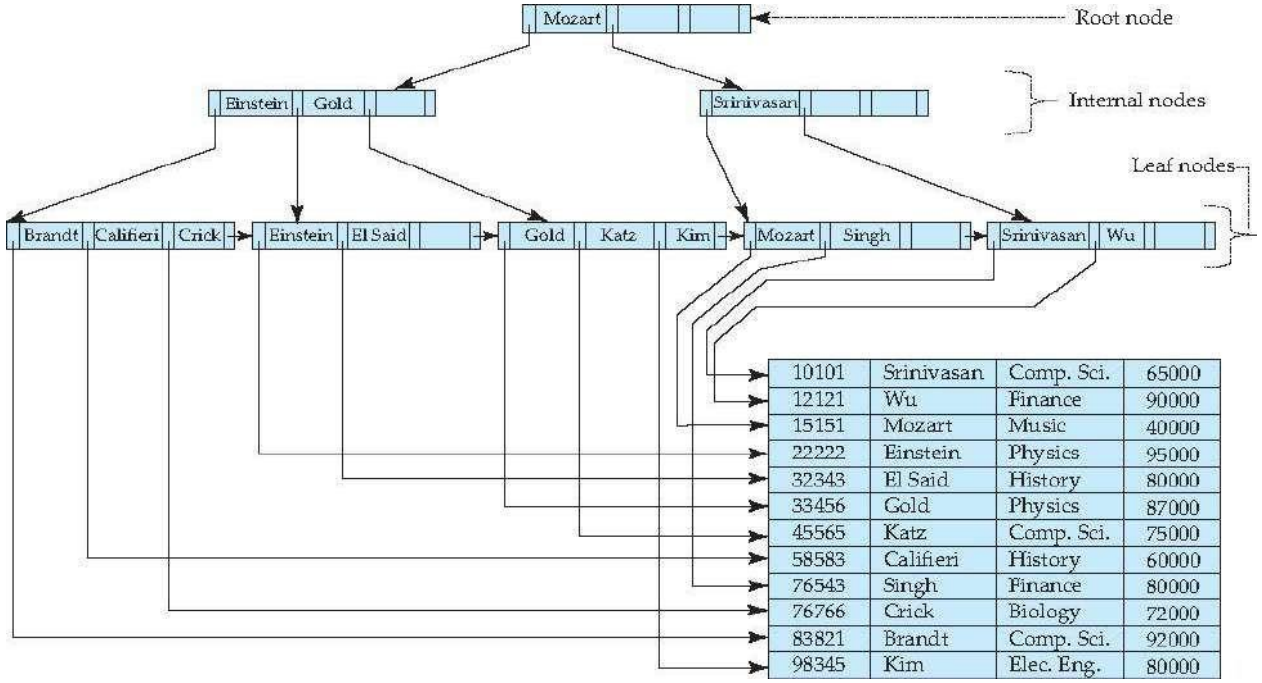
- Extra insertion and deletion overhead, space overhead.

Advantages of B+ trees outweigh disadvantages

- B+ trees are used extensively

B+ tree is a rooted tree satisfying the following properties:

1. All paths from root to leaf are of the same length
2. Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
3. A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
4. Special cases:
   a) If the root is not a leaf, it has at least 2 children.
   b) If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n-1)$ values.



B+ Tree Node Structure

Typical node

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

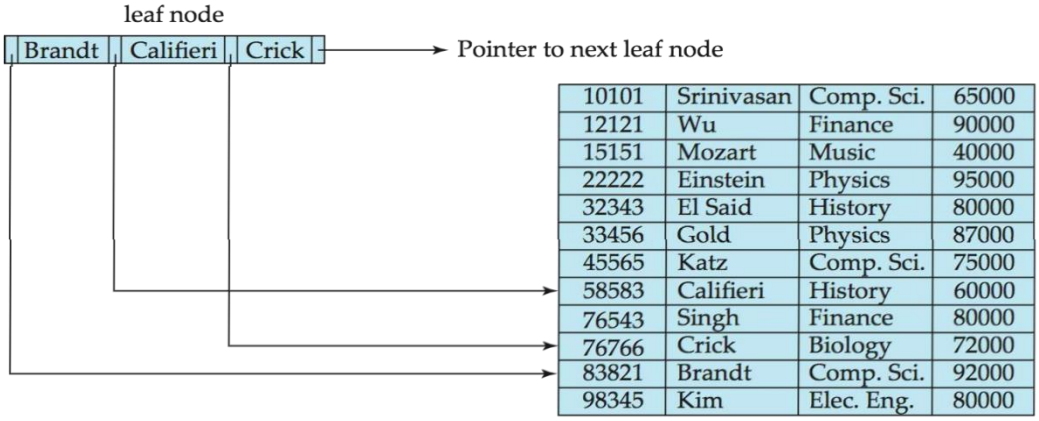
The search-keys in a node are ordered

$$K1 < K2 < K3 < \ldots < Kn-1$$

Leaf Nodes in B+ Trees

Properties of a leaf node:

1) For $i = 1, 2, \ldots, n-1$, pointer *Pi* either points to a file record with search-key value *Ki*, or to a bucket of pointers to file records, each record having search-key value *Ki.* Only need bucket structure if search-key does not form a primary key.
2) If *Li, Lj* are leaf nodes and $i < j$, *Li*'s search-key values are less than *Lj*'s search-key values
3) *Pn* points to next leaf node in search-key order



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

Non-Leaf Nodes in B+ Trees

Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:

a) All the search-keys in the sub tree to which *P*1 points are less than K_1
b) For $2 \le i \le n - 1$, all the search-keys in the sub tree to which *Pi* points have values greater than or equal to K_{i-1} and less than K_i
c) All search-keys in the sub tree to which *Pn* points have values greater than or equal to K_{n-1}

Example of a B+ tree

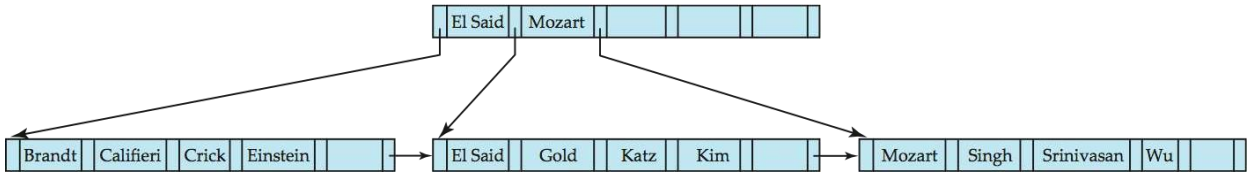


Fig: B+ tree for *instructor* file ($n = 6$)

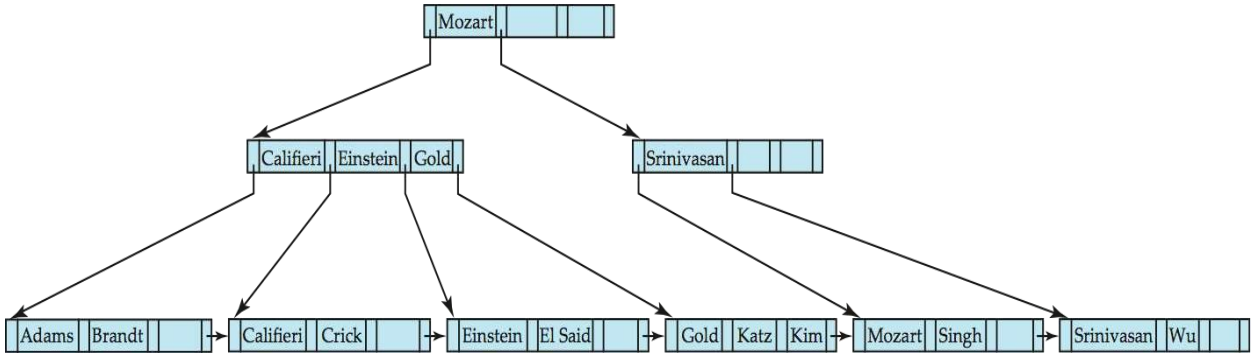
- Leaf nodes must have between 3 and 4 values ($\lceil (n-1)/2 \rceil$ and $n - 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.

Observations about B+ trees

1. Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
2. The non-leaf levels of the B+ tree form a hierarchy of sparse indices.
3. The B+ tree contains a relatively small number of levels
   i. Level below root has at least $2 * \lceil n/2 \rceil$ values
   ii. Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ values
   a. If there are *K* search-key values in the file, tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
   b. Thus searches can be conducted efficiently.
4. Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time.

Queries on B+ trees

- Find record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value s.t. $V \leq K_i$.
     2. If no such exists, set *C = last non-null pointer in C*
     3. Else { if (V= K_i) Set C = P_{i+1} else set *C* = P_i}
     }
  1. Let *i* be least value s.t. $K_i = V$
  2. If there is such a value *i,* follow pointer P_i to the desired record.
  3. Else no record with search-key value *k* exists.



- If there are *K* search-key values in the file, the height of the tree is no more than $\lceil \log\lceil n/2 \rceil (K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and *n* is typically around 100 (40 bytes per index entry).
- With 1 million search key values and *n* = 100
  - at most *log*50(1,000,000) = 4 nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

<u>Handling Duplicates</u>

With duplicate search keys

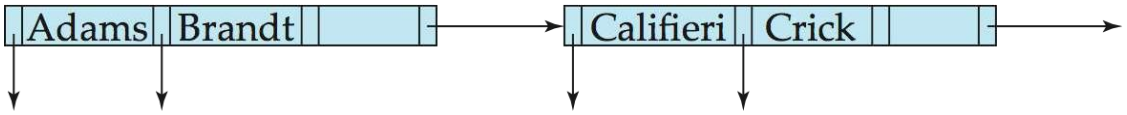
- In both leaf and internal nodes,
  - we cannot guarantee that $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
  - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
- Search-keys in the subtree to which P_i points
  - are $\leq K_{i,}$, but not necessarily $< K_{i,}$
  - To see why, suppose same search key value *V* is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V

We modify find procedure as follows

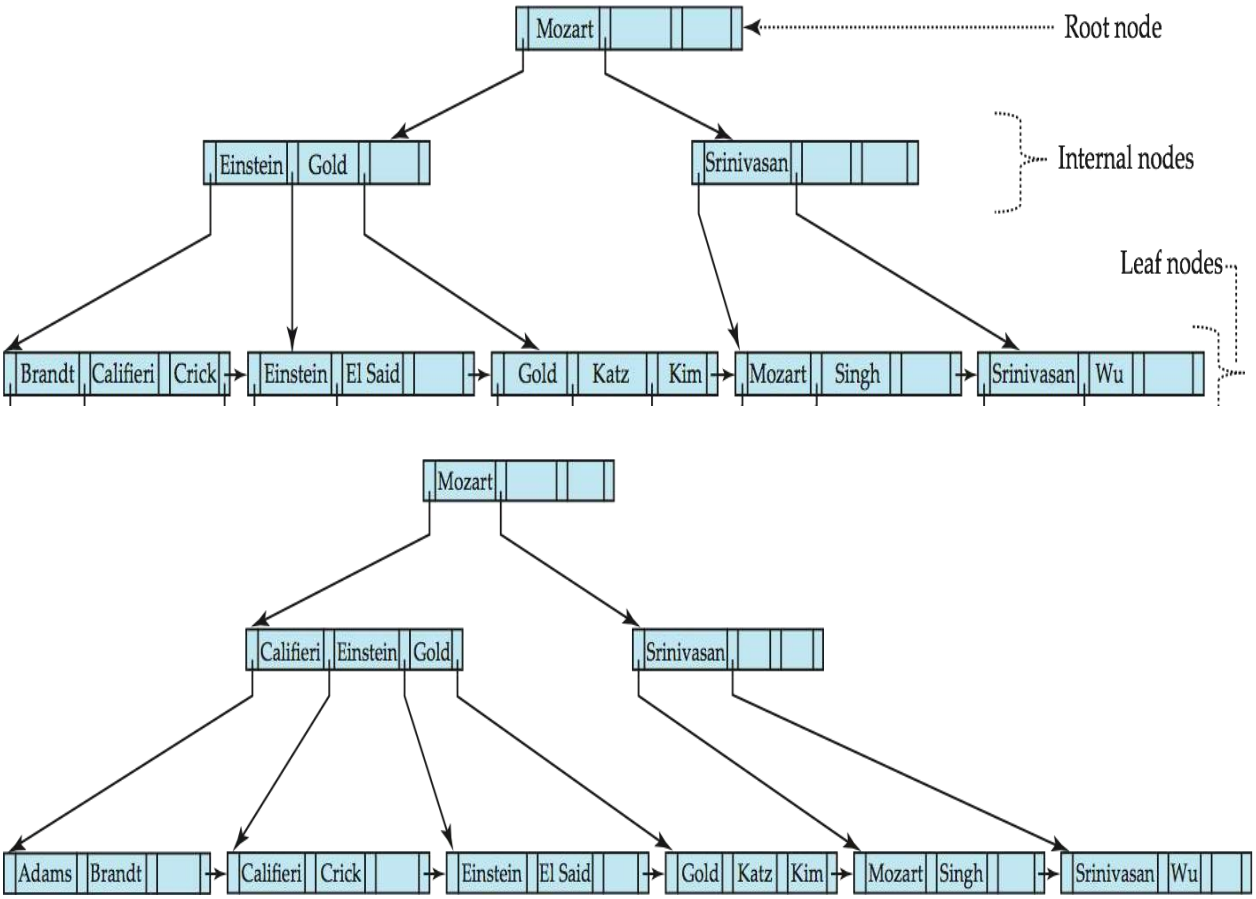
- traverse *Pi* even if *V* = *K*i
- As soon as we reach a leaf node *C* check if *C* has only search key values less than *V*
  if so set *C* = right sibling of *C* before checking whether *C* contains *V*
  Procedure printAll
- uses modified find procedure to find first occurrence of *V*
- Traverse through consecutive leaves to find all occurrences of *V*

Updates on B+ trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    3. Otherwise, split the node (along with the new (key-value, pointer) entry)
4. Splitting a leaf node:
    1. take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
    2. let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
    3. If the parent is full, split it and **propagate** the split further up.
5. Splitting of nodes proceeds upwards till a node that is not full is found.
    1. In the worst case the root node may be split increasing the height of the tree by 1
    2.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

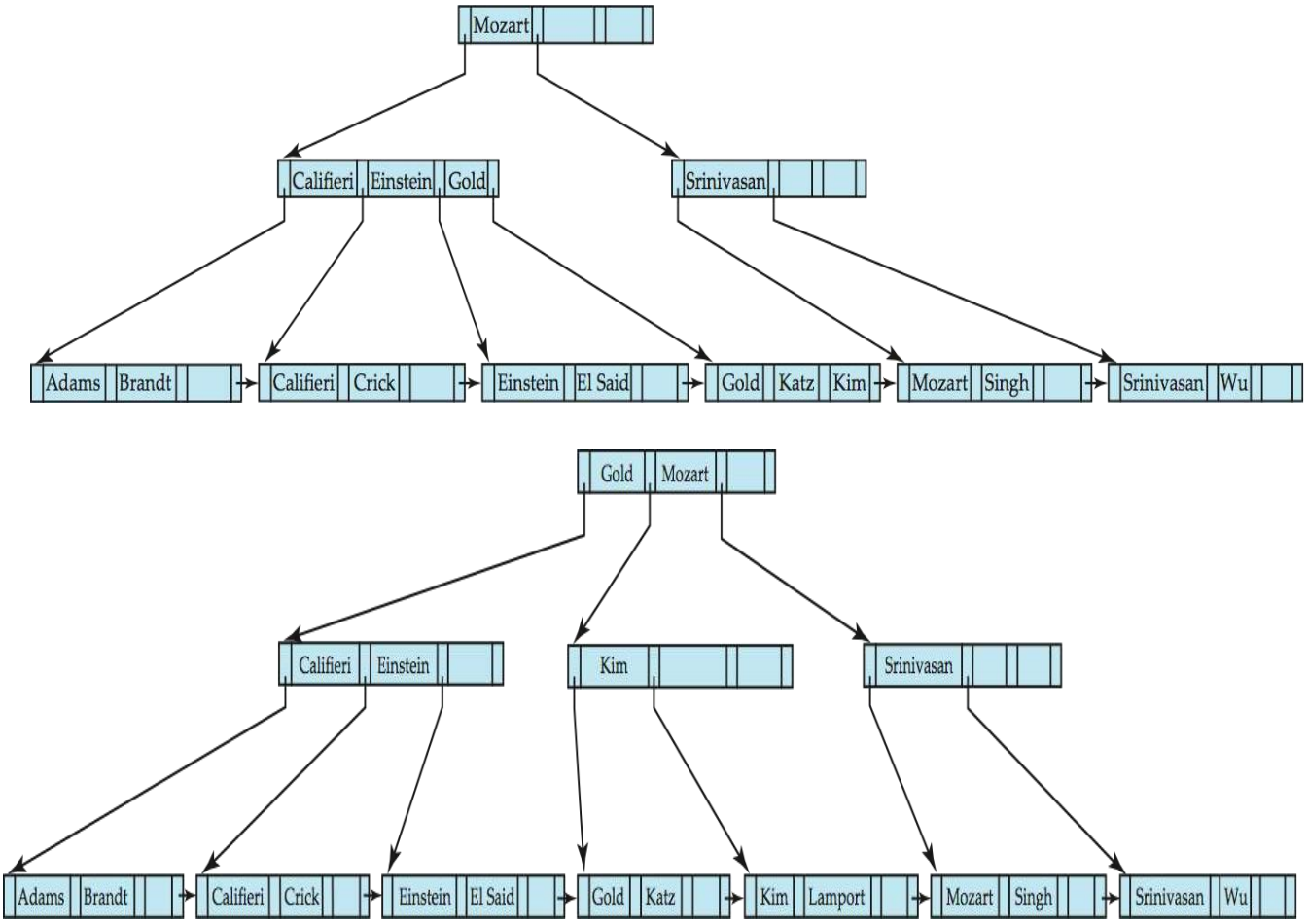


B+ tree before and after insertion of "Adams"

pseudocode from book is given below!

```
procedure insert(value K, pointer P)
if (tree is empty) create an empty leaf node L, which is also the root
else Find the leaf node L that should contain key value K

if (L has less than n − 1 key values)
    then insert in leaf (L, K, P)
    else begin /* L has n − 1 key values already, split it */
Create node L1
Copy L.P1 . . . L.Kn−1 to a block of memory T that can hold n (pointer, key-value) pairs
insert in leaf (T, K, P)
Set L1.Pn = L.Pn; Set L.Pn = L1
Erase L.P1 through L.Kn− 1 from L
Copy T.P1 through T.K_n/2_ from T into L starting at L.P1
Copy T.P_n/2_+1 through T.Kn from T into L_ starting at L1.P1
Let K1 be the smallest key-value in L1
insert in parent(L, K1, L1)
end

procedure insert in leaf (node L, value K, pointer P)
if (K < L.K1)
then insert P, K into L just before L.P1
else begin
Let Ki be the highest value in L that is less than K
Insert P, K into L just after T.Ki
end

procedure insert in parent(node N, value K_, node N_)
if (N is the root of the tree)
then begin
Create a new node R containing N, K1, N1 /* N and N1 are pointers */
Make R the root of the tree
return
 end

Let P = parent (N)
if (P has less than n pointers)
then insert (K1, N1) in P just after N
else begin /* Split P */
Copy P to a block of memory T that can hold P and (K1, N1)
Insert (K1, N1) into T just after N
Erase all entries from P; Create node P_
Copy T.P1 . . . T.P_n/2_ into P

Let K" = T.K|n/2|

Copy T.P|n/2|+1 . . . T.Pn+1 into P1
  insert in parent(P, K", P1)
   end
```

Figure: Insertion of entry in a B+ tree.



B+ tree before and after insertion of "Lamport"

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy P1,K1, …, K$\lceil n/2 \rceil$-1,P$\lceil n/2 \rceil$ from M back into node N
  - Copy P$\lceil n/2 \rceil$+1,K$\lceil n/2 \rceil$+1,…,Kn,Pn+1 from M into newly allocated node N'
  - Insert (K$\lceil n/2 \rceil$,N') into parent N

Updates on B+ trees: Deletion

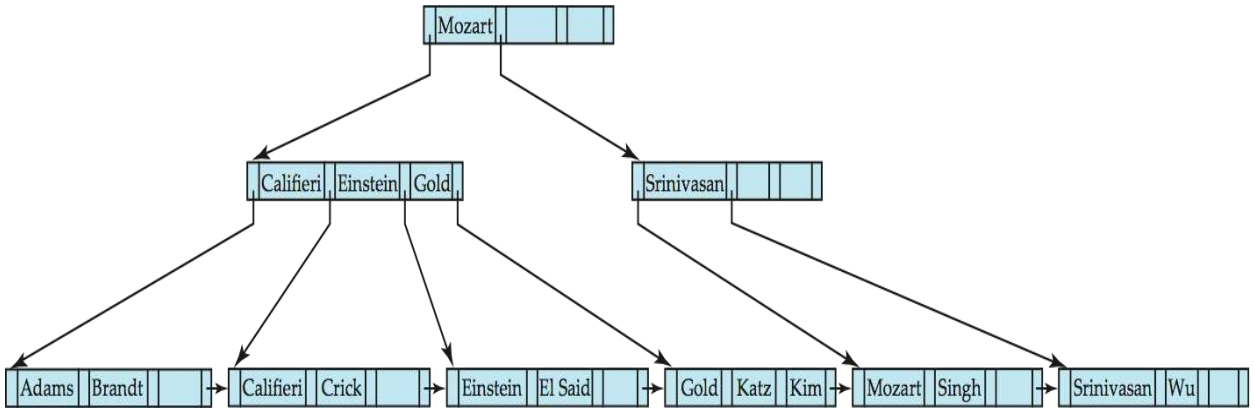
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (*Ki*–1, *Pi),* where *Pi* is the pointer to the deleted node, from its parent, recursively using the above procedure.

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:

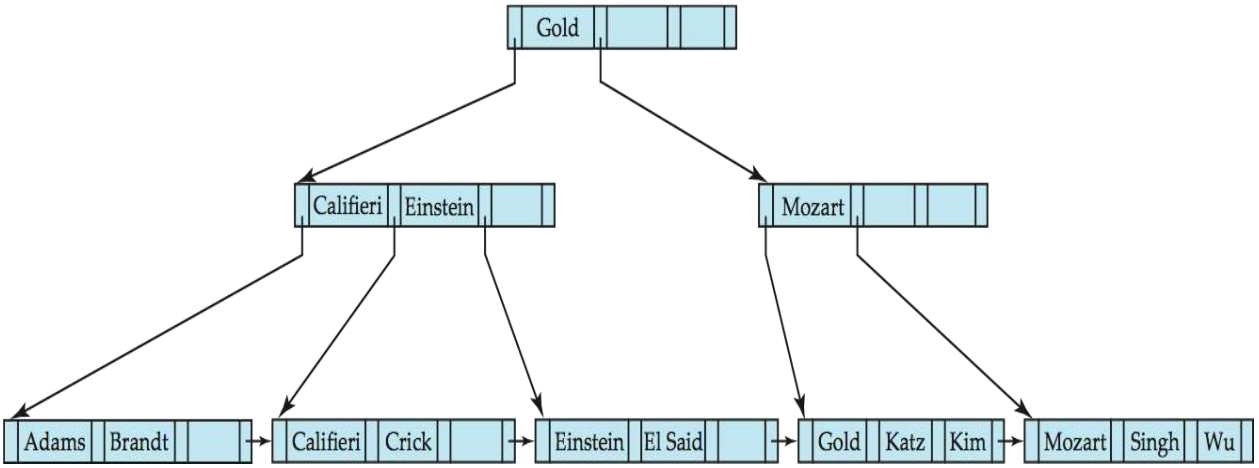
- Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
- Update the corresponding search-key value in the parent of the node.

- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

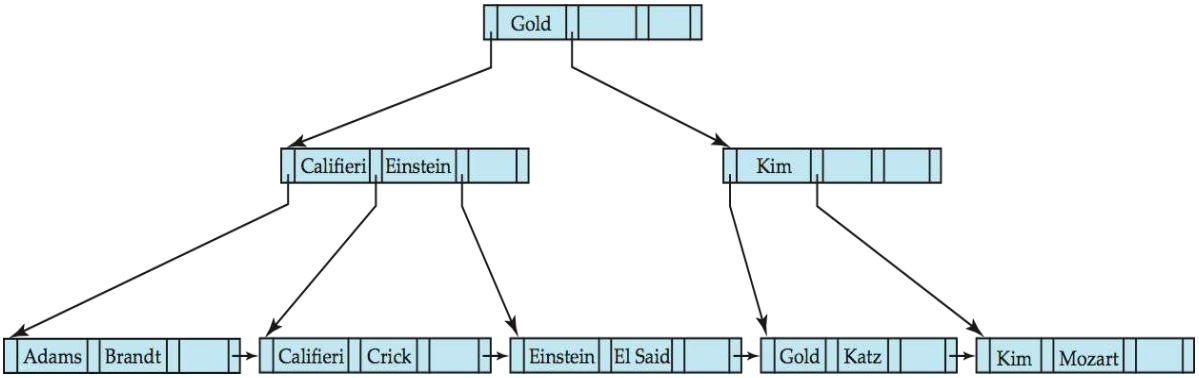
Examples of B+ tree Deletion



Before and after deleting "Srinivasan"

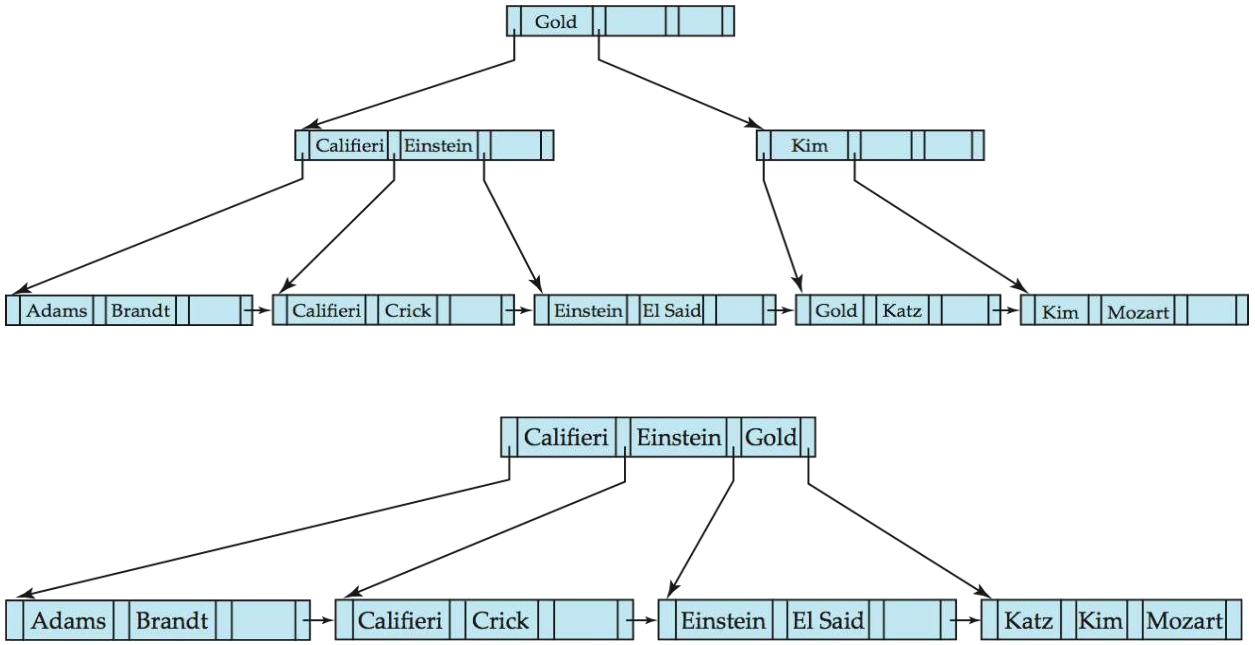


Deleting "Srinivasan" causes merging of under-full leaves.leaf node can become empty only for n=3!



Deletion of "Singh" and "Wu" from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result



Before and after deletion of "Gold" from earlier example

- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

B+ tree File Organization

- Index file degradation problem is solved by using B+ tree indices.
- Data file degradation problem is solved by using B+ tree File Organization.
- The leaf nodes in a B+ tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B+ tree index.

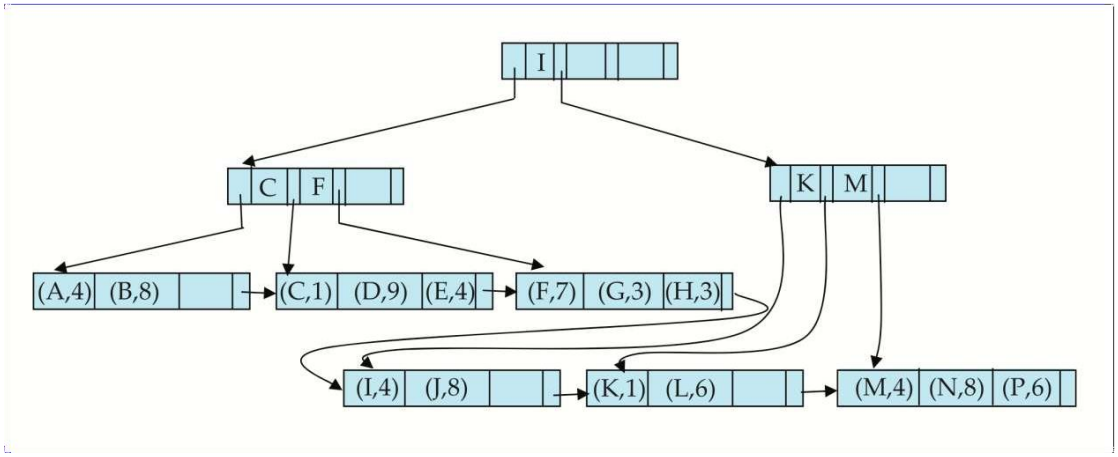


Fig: Example of B+ tree File Organization

- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Other Issues in Indexing

- **Record relocation and secondary indices**
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B+ tree file organizations become very expensive
  - *Solution*: use primary-index search key instead of record pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique

Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the sub trees separated by the key value
      - EX: "Silas" and "Silberschatz" can be separated by "Silb"
  - Keys in leaf node can be compressed by sharing common prefixes

Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B+ tree requires ≥ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms)
  - insert in sorted order
    - insertion will go to existing page (or cause a split)
    - much improved IO performance, but most leaf nodes half full

- Efficient alternative 2: **Bottom-up B+ tree construction**
  - As before sort entries
  - And then create tree layer-by-layer, starting with leaf level
    - details as an exercise
  - Implemented as part of bulk-load utility by most database systems

B-Tree Index Files

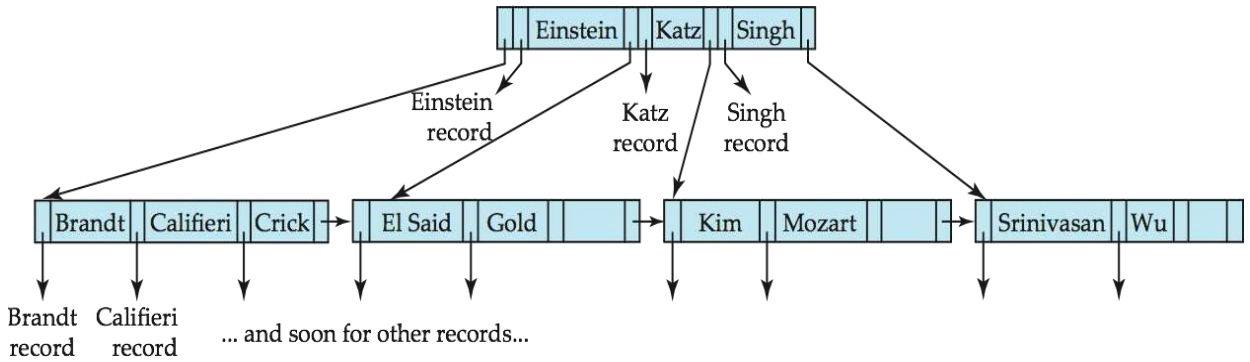
- Similar to B+ tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

(a)

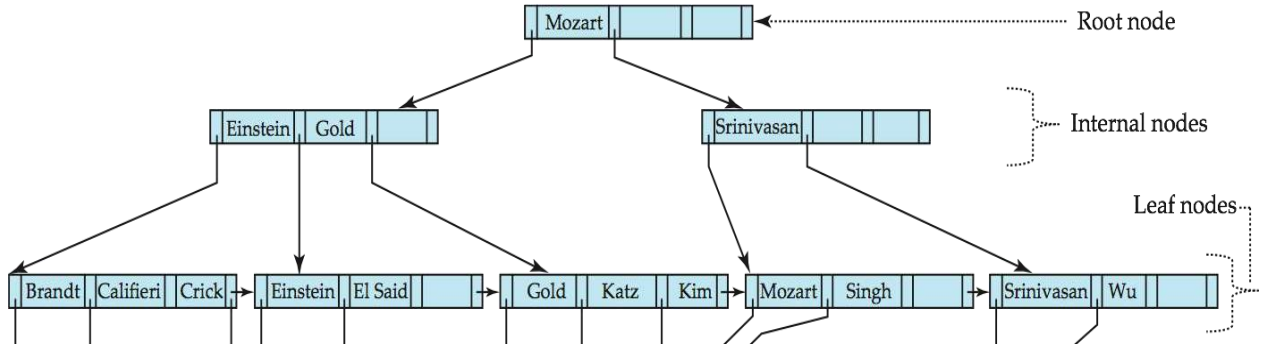
(b)

Nonleaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+ tree (below) on same data



Advantages of B-Tree indices:

- May use less tree nodes than a corresponding B+ tree.
- Sometimes possible to find search-key value before reaching leaf node.

Disadvantages of B-Tree indices:

- Only small fraction of all search-key values are found early
- Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B+ tree
- Insertion and deletion more complicated than in B+ trees
- Implementation is harder than B+ trees.

Typically, advantages of B-Trees do not outweigh disadvantages.

Multiple-Key Access

Use multiple indices for certain types of queries.

Example:
select *ID*
from *instructor*
where *dept_name* = "Finance" **and** *salary* = 80000

Possible strategies for processing query using indices on single attributes:

1. Use index on *dept_name* to find instructors with department name Finance; test *salary = 80000*
2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = "Finance".
3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department.

Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Indices on Multiple Keys

Composite search keys are search keys containing more than one attribute
EX: (*branch_name, balance*)
Lexicographic ordering: (a1, a2) < (b1, b2) if either

1. a1 < b1, or
2. a1=b1 and a2 < b2

Indices on Multiple Attributes

Suppose we have an index on combined search-key
(dept_*name, salary*).

With the **where** clause
where *dept_name* = "Finance" **and** *salary* = 80000

the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.

- Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.

- Can also efficiently handle
  where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
  where *dept_name* < "Finance" **and** *balance* = 80000
  - May fetch many records that satisfy the first but not the second condition

Non-Unique Search Keys

Alternatives:

- Buckets on separate block (bad idea)
- List of tuple pointers with each key
  - Low space overhead, no extra cost for queries
  - Extra code to handle read/update of long lists
  - Deletion of a tuple can be expensive if there are many duplicates on search key
- Make search key unique by adding a record-identifier
  - Extra storage overhead for keys
  - Simpler code for insertion/deletion
  - Widely used

Other Issues in Indexing

Covering indices

Add extra attributes to index so (some) queries can avoid fetching the actual records

- Particularly useful for secondary indices
  - Why?

Can store extra attributes only at leaf

Record relocation and secondary indices

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B+ tree file organizations become very expensive
- *Solution*: use primary-index search key instead of record pointer in secondary index
  - Extra traversal of primary index to locate record
    - Higher cost for queries, but node splits are cheap
  - Add record-id if primary-index search key is non-unique

Hashing

Static Hashing

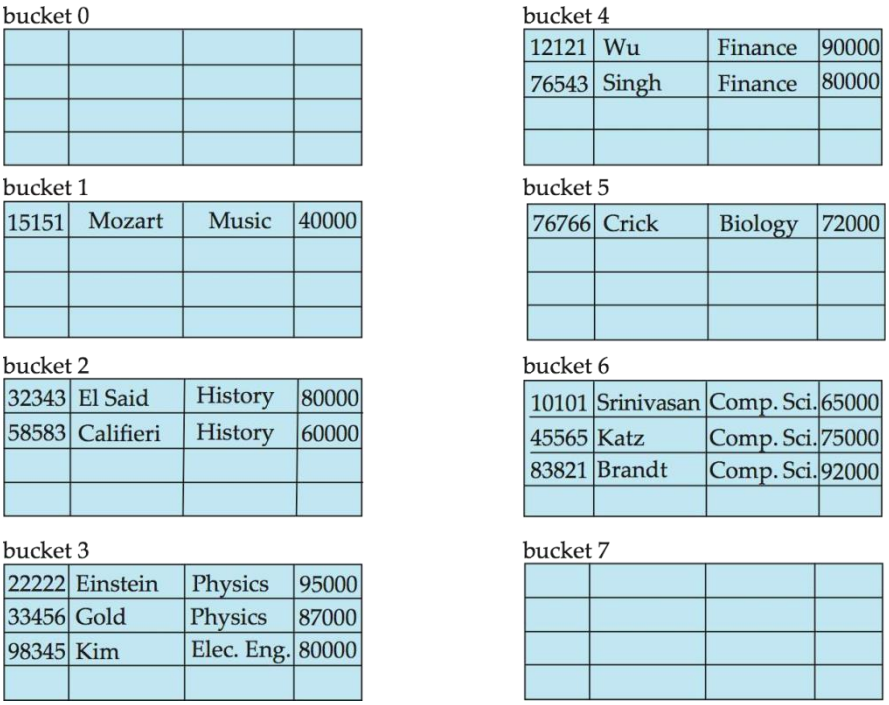
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key

There are 10 buckets,

- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10.
  EX: h(Music) = 1 h(History) = 2
  h(Physics) = 3 h(Elec. Eng.) = 3



Hash file organization of *instructor* file, using *dept_name* as key

Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on internal binary representation of search-key.
    - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

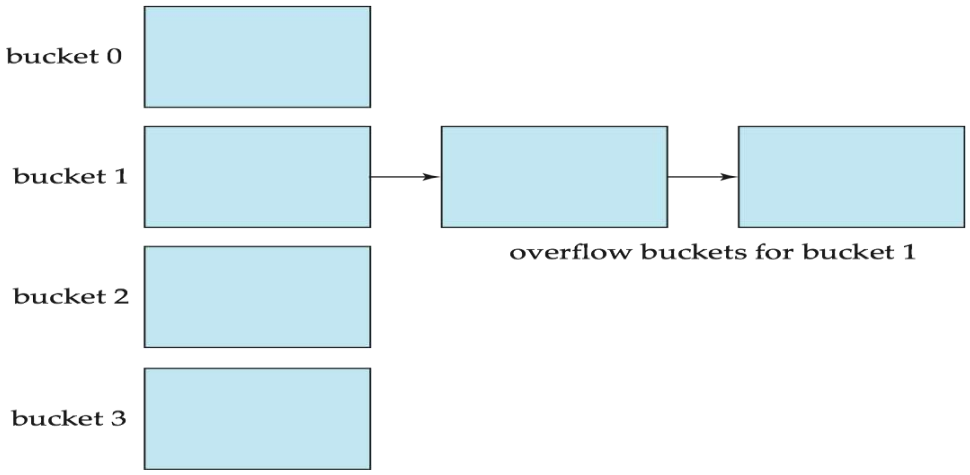
Handling of Bucket Overflows: Bucket overflow can occur because of

1. Insufficient buckets
2. Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets.*

Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list. Above scheme is called ***closed hashing*.**

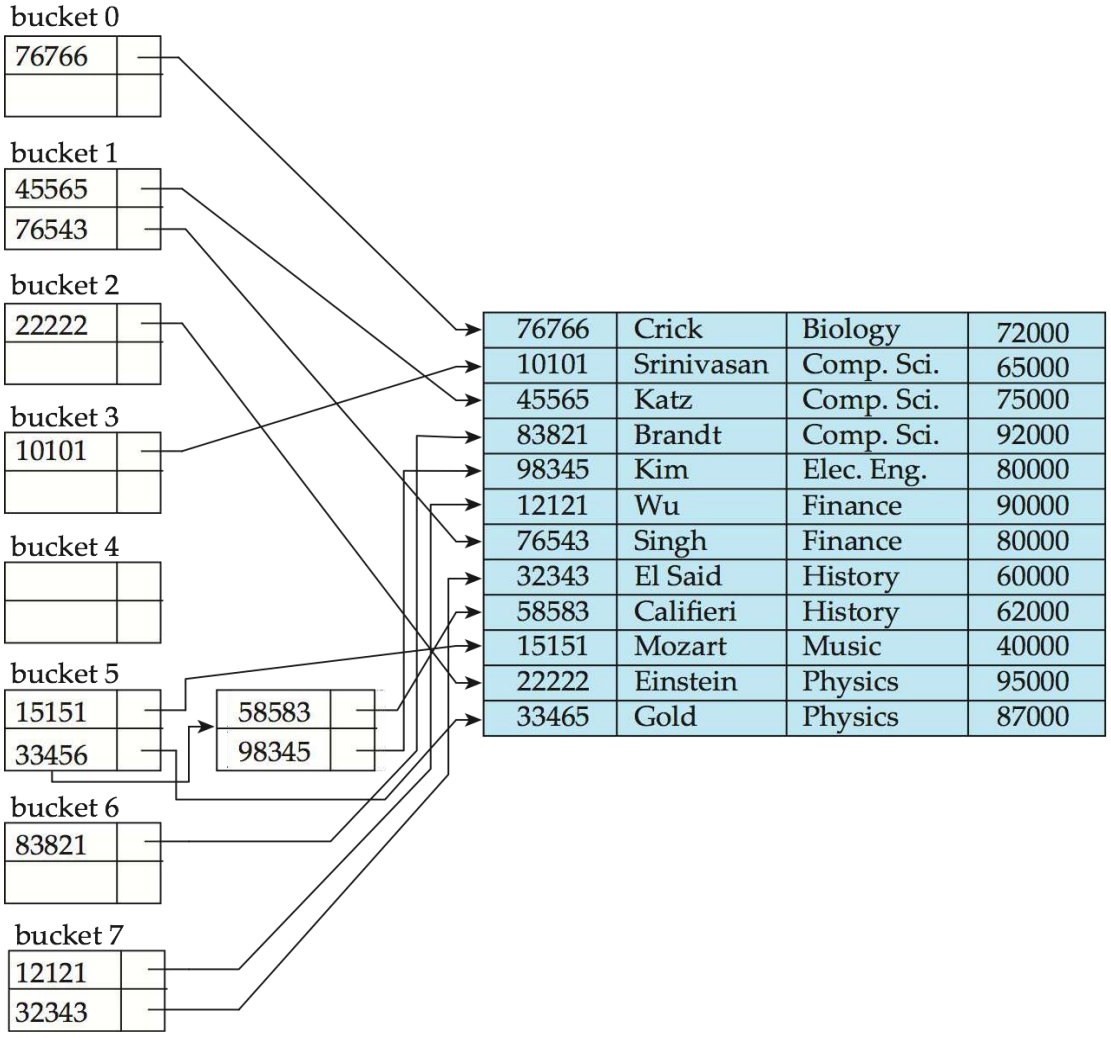
- An alternative, called ***open hashing***, which does not use overflow buckets, is not suitable for database applications.



Hash Indices: Hashing can be used not only for file organization, but also for index-structure creation.

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index: Hash index on *instructor,* on attribute *ID*



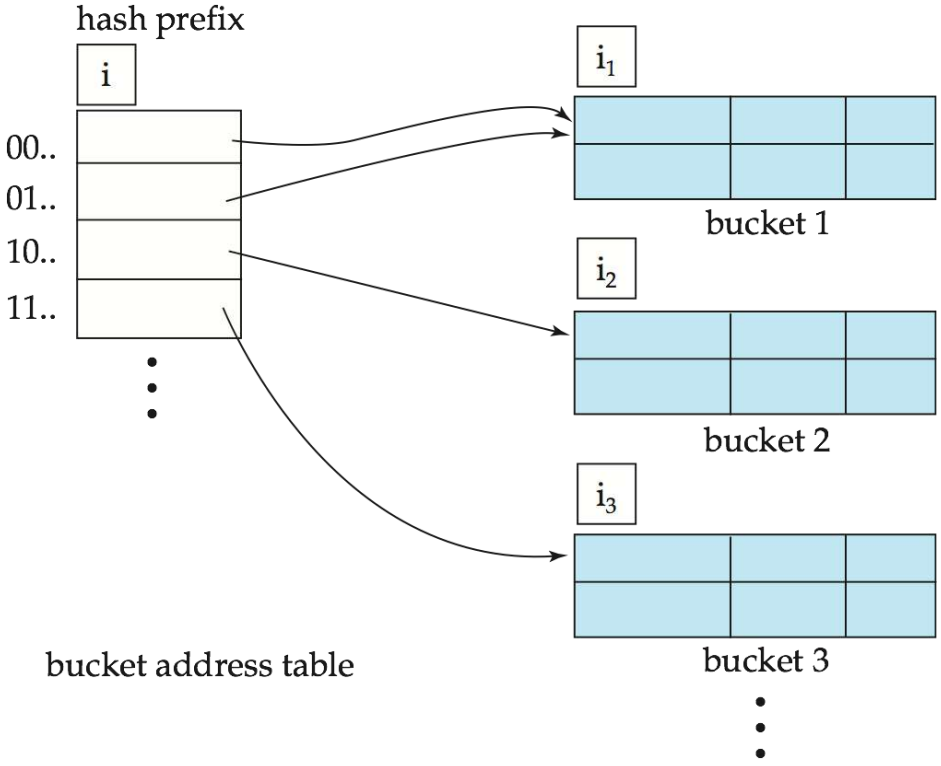
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range, typically *b*-bit integers, with *b*= 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \le i \le 32$.
  - Bucket address table size = 2i. Initially $i = 0$
  - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket
  - Thus, actual number of buckets is < 2*i*
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, *i*2 = *i*3 = *i*, whereas *i*1 = *i* – 1

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first *ij* bits.
- To locate the bucket containing search-key K_j:
  - Compute *h(Kj) = X*
  - Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value *Kj*
  - follow same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j* insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted.
    - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure:

To split a bucket *j* when inserting record with search-key value *Kj*:

1. If *i* > *ij* (more than one pointer to bucket *j*)
   a. allocate a new bucket *z*, and set *ij* = *iz* = (*ij* + 1)
   b. Update second half of bucket address table entries originally pointing to *j,* to point to *z*
   c. remove each record in bucket *j* and reinsert (in *j* or *z*)
   d. recompute new bucket for *Kj* and insert record in the bucket (further splitting is required if the bucket is still full)
2. If *i* = *ij* (only one pointer to bucket *j*)
   a. If *i* reaches some limit *b*, or too many splits have happened in this insertion, create an overflow bucket
   b. Else
      i. increment *i* and double the size of the bucket address table.
      ii. replace each entry in the table by two entries that point to the same bucket.
      iii. recompute new bucket address table entry for K_j
      Now *i* > i_j so use the first case above.

Deletion in Extendable Hash Structure:

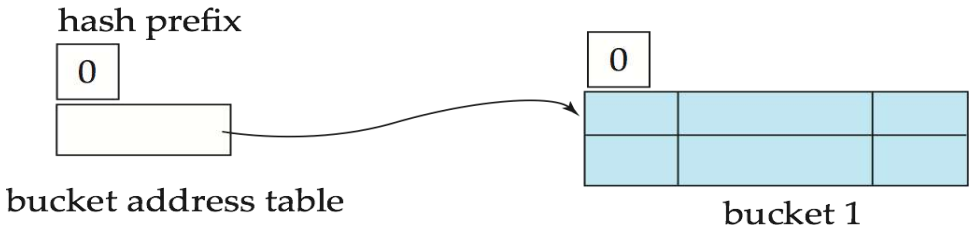
To delete a key value,

- locate it in its bucket and remove it.
- The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
- Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of ij and same ij –1 prefix, if it is present)
- Decreasing bucket address table size is also possible
  - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

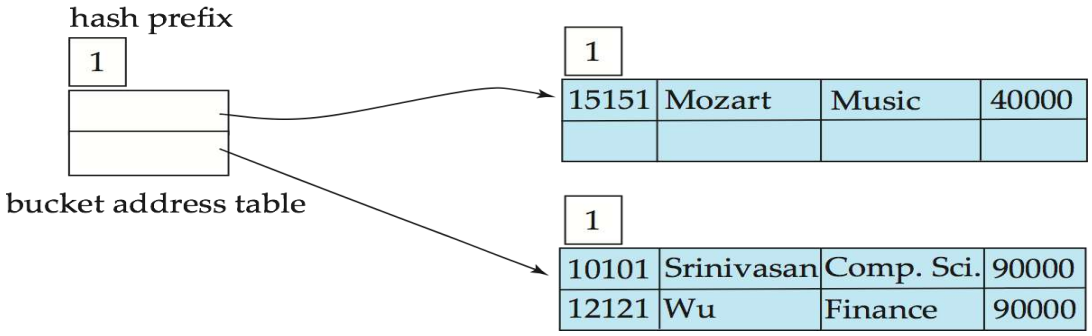
Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

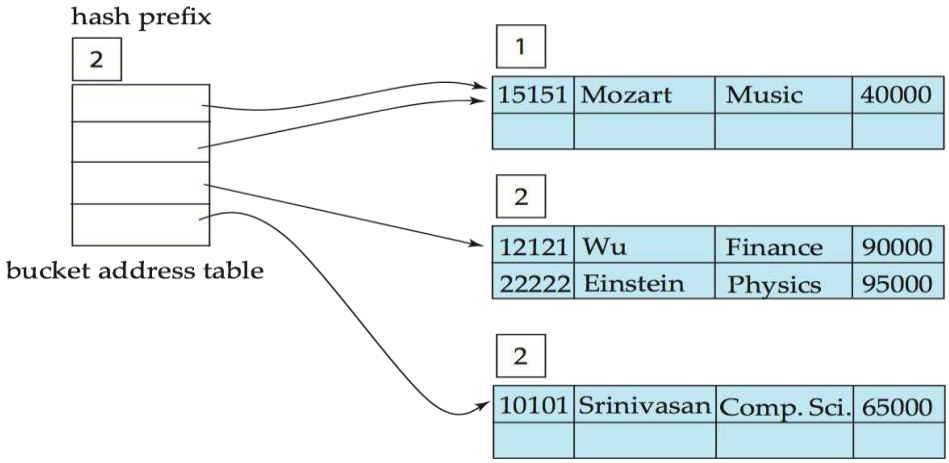
Initial Hash structure, bucket size = 2



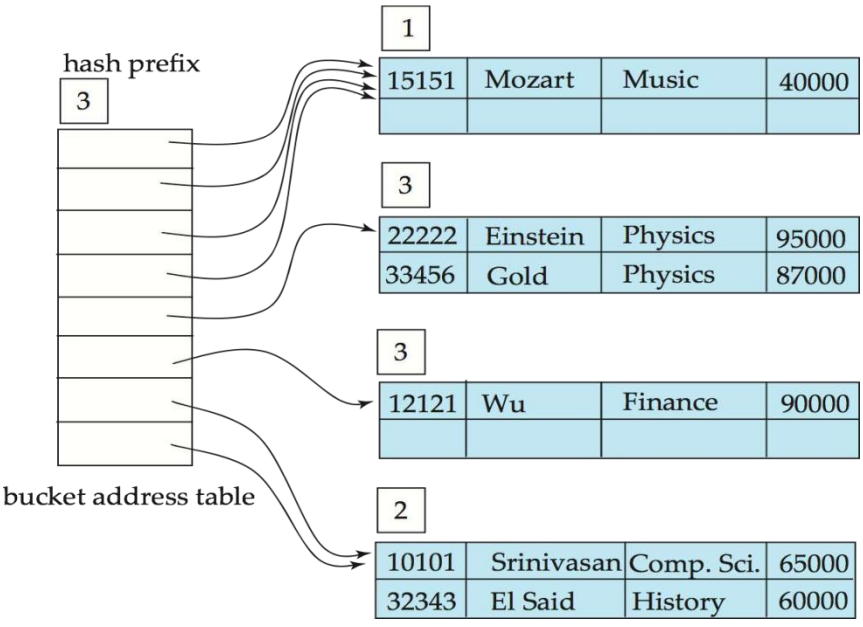
Hash structure after insertion of "Mozart", "Srinivasan", and "Wu" records



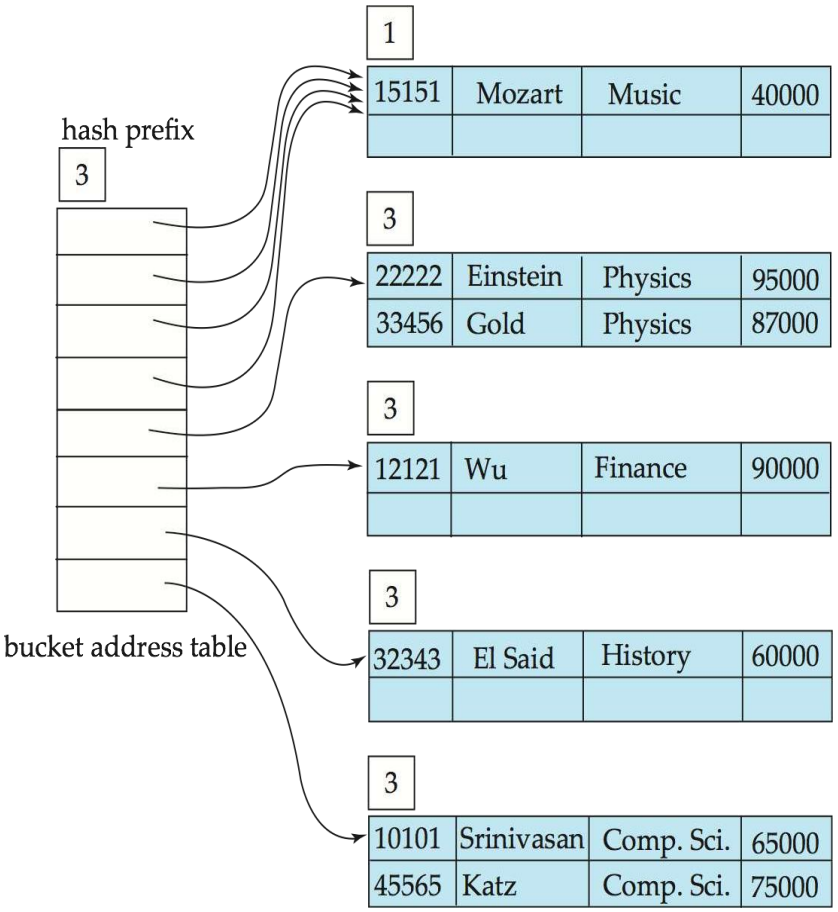
Hash structure after insertion of Einstein record



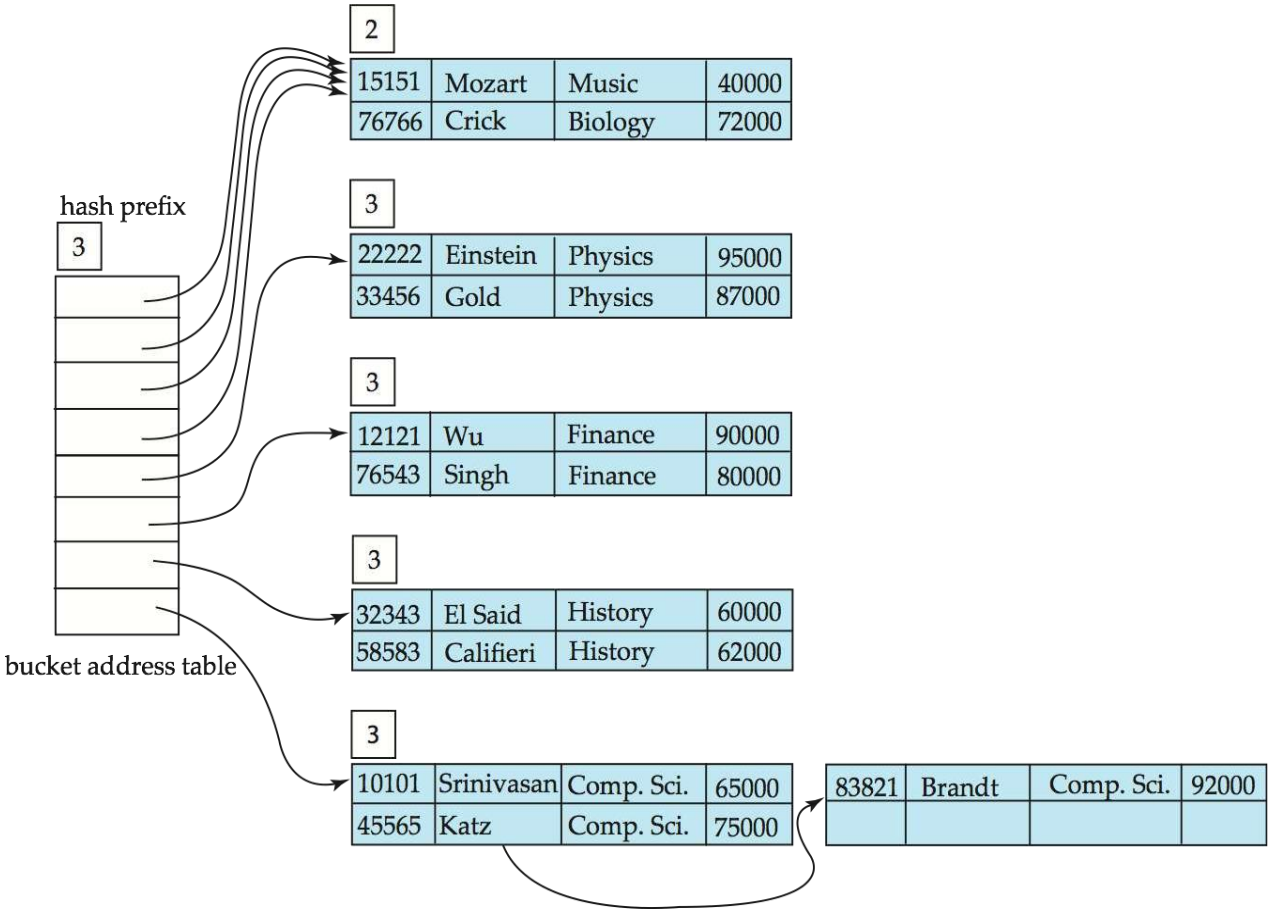
Hash structure after insertion of Gold and El Said records



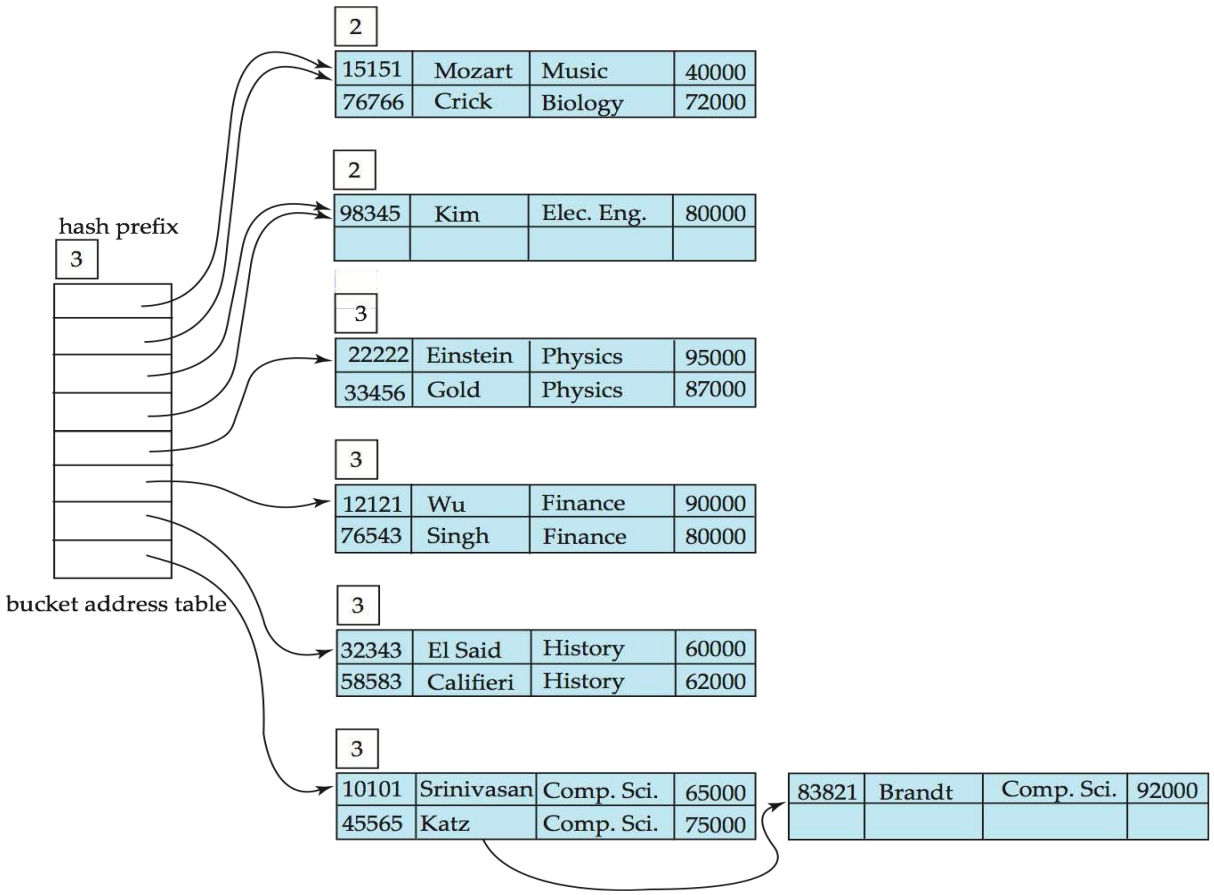
Hash structure after insertion of Katz record



And after insertion of eleven records



And after insertion of Kim record in previous hash structure



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B+ tree file organization to store bucket address table
  - Changing size of bucket address table is an expensive operation
- **Linear hashing** is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B+ trees

Bitmap Indices: Bitmap indices are a special type of index designed for efficient querying on multiple keys

- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - EX: gender, country, state, …
  - EX: income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits
- In its simplest form a bitmap index on an attribute has a bitmap for each value of attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - EX: 100110 AND 110011 = 100010
    100110 OR 110011 = 110111
    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster
- Bitmap indices generally very small compared with relation size
  - EX: if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
  - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - **Existence bitmap** to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A=v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A=v*):
    - intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - EX: 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B+ trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B+ tree indices

Index Definition in SQL

Create an index

create index <index-name> **on** <relation-name>
(<attribute-list>)

EX:: **create index** *b-index* **on** *branch(branch_name)*

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.

- Not really required if SQL **unique** integrity constraint is supported

To drop an index

drop index <index-name>

Most database systems allow specification of type of index, and clustering.

TRANSACTIONS

Transaction Concept: A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

EX: transaction to transfer $50 from account A to account B:

read(*A*) *A* := *A* – 50 **write**(*A*) **read**(*B*) *B* := *B* + 50 **write**(*B)*

Two main issues to deal with:

- Failures of various kinds, such as hardware failures and system crashes
- Concurrent execution of multiple transactions

Atomicity requirement — If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state

- Failure could be due to software or hardware

✓ System should ensure that updates of a partially executed transaction are not reflected in database

Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Consistency requirement in above example: the sum of A and B is unchanged by the execution of the transaction. In general, consistency requirements include

- Explicitly specified integrity constraints such as primary keys and foreign keys
- Implicit integrity constraints

– EX: sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand

- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Isolation requirement — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

	T1	**T2**
1.	**read**(*A*)	
2.	*A* := *A* – 50	
3.	**write**(*A*)	
		read(A), read(B), print(A+B)
4.	**read**(*B*)	
5.	*B* := *B* + 50	
6.	**write**(*B*	

Isolation can be ensured trivially by running transactions **serially**

-that is, one after the other.

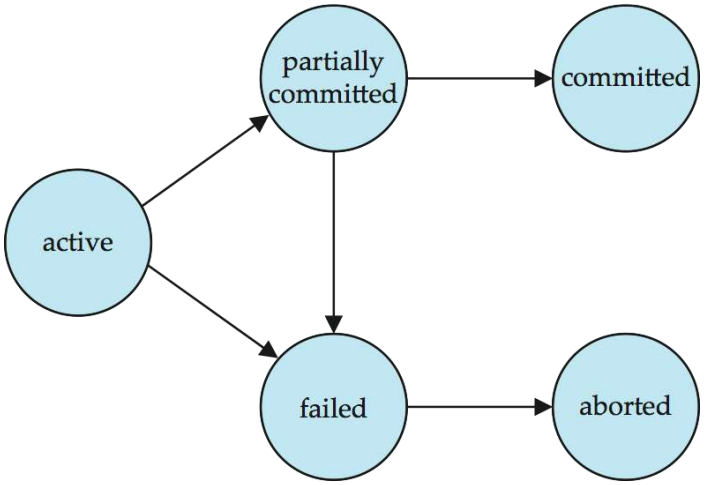
However, executing multiple transactions concurrently has significant benefits.

ACID Properties: A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

1. **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
2. **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
3. **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
   a. That is, for every pair of transactions *Ti* and *Tj,* it appears to *Ti* that either *Tj,* finished execution before *Ti* started, or *Tj* started execution after *Ti* finished.
4. **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State:

1. **Active** – the initial state; the transaction stays in this state while it is executing
2. **Partially committed** – after the final statement has been executed.
3. **Failed** -- after the discovery that normal execution can no longer proceed.
4. **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
   a. restart the transaction
      i. can be done only if no internal logical error
   b. kill the transaction
5. **Committed** – after successful completion.



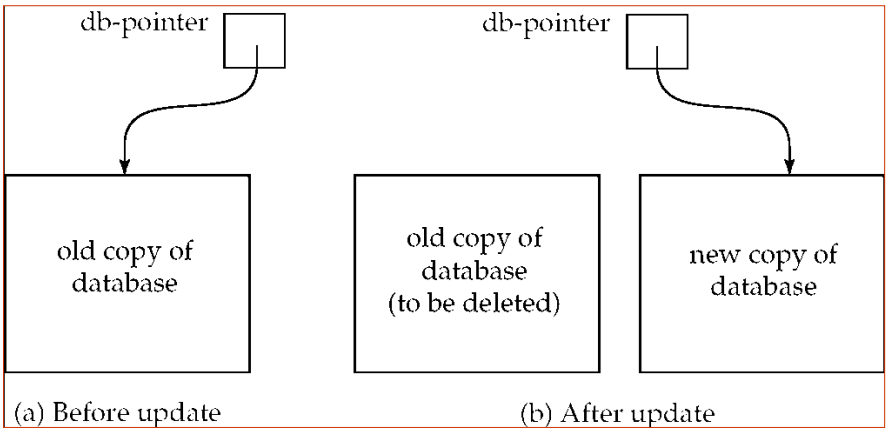
Implementation of Atomicity and Durability: The **recovery-management** component of a database system implements the support for atomicity and durability.

EX: the ***shadow-database*** scheme:

-all updates are made on a *shadow copy* of the database

db_pointer is made to point to the updated shadow copy after

- the transaction reaches partial commit and
- all updated pages have been flushed to disk.



db_pointer always points to the current consistent copy of the database.

-- In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:

- Assumes that only one transaction is active at a time.
- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases
    - Variant called shadow paging reduces copying of data, but is still not practical for large databases
- Does not handle concurrent transactions

Concurrent Executions: Multiple transactions are allowed to run concurrently in the system. Advantages are:

- increased processor and disk utilization, leading to better transaction *throughput*

- EX: one transaction can be using the CPU while another is reading from or writing to the disk

- reduced average response time for transactions: short transactions need not wait behind long ones.

Concurrency control schemes – These are the mechanisms to achieve isolation. That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules: A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed. a schedule for a set of transactions must consist of all instructions of those transactions and must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have commit instructions as the last statement. by default transaction assumed to execute commit instruction as its last step. A transaction that fails to successfully complete its execution will have an abort instruction as the last statement.

Schedule 1: Let *T*1 transfer $50 from *A* to *B*, and *T*2 transfer 10% of the balance from *A* to *B.*

A serial schedule in which *T*1 is followed by *T*2:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2:

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedule 3:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Let *T*1 and *T*2 be the transactions defined previously. The above schedule is not a serial schedule, but it is *equivalent* to Schedule 1.
Note: In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4: The following concurrent schedule does not preserve the value of (*A* + *B)*.

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
1. conflict serializability
2. view serializability

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

Instructions *li* and *lj* of transactions *Ti* and *Tj* respectively, **conflict** if and only if there exists some item *Q* accessed by both *li* and *lj*, and at least one of these instructions wrote *Q.*

1. *li* = **read**(*Q), lj* = **read**(*Q*). *li* and *lj* don't conflict.
2. *li* = **read**(*Q), lj* = **write**(*Q*). They conflict.
3. *li* = **write**(*Q), lj* = **read**(*Q*). They conflict
4. *li* = **write**(*Q), lj* = **write**(*Q*). They conflict

Intuitively, a conflict between *li* and *lj* forces a (logical) temporal order between them. If *li* and *lj* are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability: If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.

We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule. Schedule 3 can be transformed into Schedule 6, a serial schedule where *T*2 follows *T*1, by series of swaps of non-conflicting instructions. Therefore Schedule 3 is conflict serializable

Schedule 3

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 6

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule < *T*3, *T*4 >, or the serial schedule < *T*4, *T*3 >.

View Serializability

Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met, for each data item *Q,*

1. If in schedule S, transaction *Ti* reads the initial value of *Q*, then in schedule *S´* also transaction *Ti* must read the initial value of *Q.*
2. If in schedule S transaction *Ti* executes **read**(*Q*), and that value was produced by transaction *Tj* (if any), then in schedule *S´* also transaction *Ti* must read the value of *Q* that was produced by the same **write**(Q) operation of transaction *Tj* .
3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S´.*

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.
A schedule *S* is **view serializable** if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable. Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

The schedule below produces same outcome as the serial schedule < $T1$, $T5$ >, yet is not conflict equivalent or view equivalent to it.

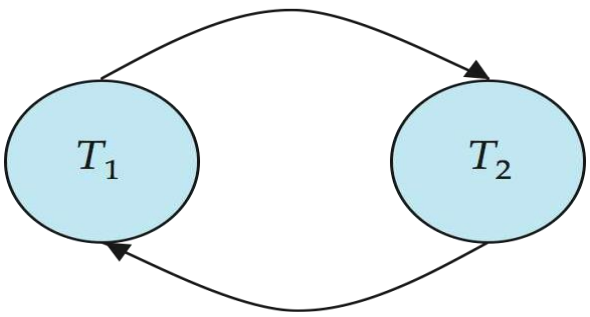
T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

Determining such equivalence requires analysis of operations other than read and write.

Testing for Serializability

- Consider some schedule of a set of transactions $T1$, $T2$, ..., Tn
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from Ti to Tj if the two transaction conflict and Ti accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Example



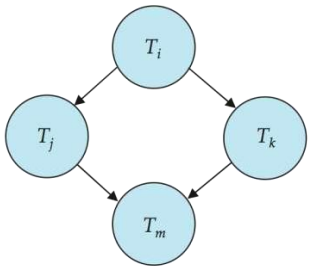
Example Schedule (Schedule A) + Precedence Graph

T1	*T2*	*T3*	*T4*	*T5*
read(Y) read(Z) read(U) read(U) write(U)	read(X) read(Y) write(Y)	 write(Z)	 read(Y) write(Y) read(Z) write(Z)	 read(V) read(W) read(W)

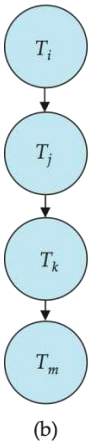
Test for Conflict Serializability

A schedule is conflict serializable if and only if its precedence graph is acyclic.

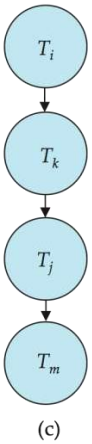
- Cycle-detection algorithms exist which take order $n2$ time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a Serializability order for Schedule A would be $T5 \rightarrow T1 \rightarrow T3 \rightarrow T2 \rightarrow T4$



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
  - Extension to test for view Serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view Serializability can still be used.

Recoverable Schedules: Need to address the effect of transaction failures on concurrently running transactions.

Recoverable schedule — if a transaction *Tj* reads a data item previously written by a transaction *Ti* , then the commit operation of *Ti* appears before the commit operation of *Tj.*

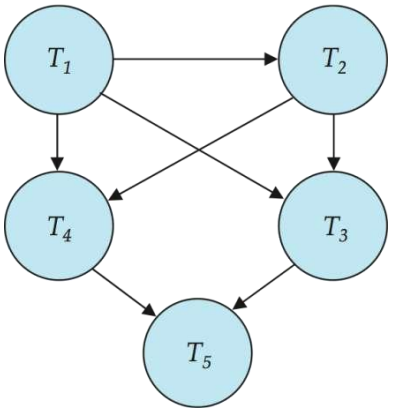
The following schedule (Schedule 11) is not recoverable if *T9* commits immediately after the read

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If *T*8 should abort, *T*9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks: A single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		



If *T*10 fails, *T*11 and *T*12 must also be rolled back.

✓ Can lead to the undoing of a significant amount of work.

Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions *Ti* and *Tj* such that *Tj* reads a data item previously written by *Ti*, the commit operation of *Ti* appears before the read operation of *Tj*.

Every cascadeless schedule is also recoverable. It is desirable to restrict the schedules to those that are cascadeless.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for Serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless.
- Concurrency control protocols generally do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for Serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

- EX: a read-only transaction that wants to get an approximate total balance of all accounts
- EX: database statistics computed for query optimization can be approximate

Such transactions need not be serializable with respect to other transactions

- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default

EX: Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL: In SQL, a transaction begins implicitly

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - EX: in JDBC, connection. setAutoCommit(false);

UNIT V

Concurrency Control: Lock-based Protocols, Timestamp-based Protocols, Validation-based Protocols, Multiple Granularity, Multi-version Schemes, Deadlock Handling, Insert and Delete Operations, Weak Levels of Consistency, Concurrency of Index Structures.

Recovery System: Failure Classification, Storage Structure, Recovery and Atomicity, Log-Based Recovery, Recovery with Concurrent Transactions, Buffer Management, Failure with Loss of Nonvolatile Storage, Advanced Recovery Techniques, Remote Backup Systems

CONCURRENCY CONTROL

LOCK-BASED PROTOCOLS:

A locking **protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

A lock is a mechanism to control concurrent access to a data item. Data items can be locked in two modes:

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix:

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

Example of a transaction performing locking:

T2: **lock-S***(A)*;
read *(A)*;
unlock*(A)*;
lock-S*(B)*;
read *(B)*;
unlock*(B)*;
display*(A+B)*

- Locking as above is not sufficient to guarantee Serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

Pitfalls of Lock-Based Protocols: Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Neither *T3* nor *T4* can make progress — executing lock**-S** *(B)* causes *T4* to wait for *T3* to release its lock on *B*, while executing lock**-X** *(A)* causes *T3* to wait for *T4* to release its lock on *A*. Such a situation is called a **deadlock**. To handle a deadlock one of *T3* or *T4* must be rolled back and its locks released. The potential for deadlock exists in most locking protocols.

- **Starvation** is also possible if concurrency control manager is badly designed.
  For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol:

This is a protocol which ensures conflict-serializable schedules.

Phase 1: Growing Phase: The transaction may obtain locks and transaction may not release locks.

Phase 2: Shrinking Phase: The transaction may release locks and transaction may not obtain locks.

- This protocol assures Serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (Ex: ordering of access to data), two-phase locking is needed for conflict Serializability in the following sense:
  Given a transaction *T*i that does not follow two-phase locking, we can find a transaction *Tj* that uses two-phase locking, and a schedule for *Ti* and *Tj* that is not conflict serializable.

Lock Conversions

- Two-phase locking with lock conversions:
  - ✓ First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)

- ✓ Second Phase:
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

- This protocol assures Serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks: A transaction *T*i issues the standard read/write instruction, without explicit locking calls.

The operation **read** (*D*) is processed as:

```
if Ti has a lock on D then
        read (D)
 else begin
        if necessary wait until no other transaction has a lock-X on D
        grant Ti a lock-S on D;
        read (D)
      end
```

write *(D)* is processed as:

```
if Ti has a lock-X on D then
    write (D)
 else begin
    if necessary wait until no other trans. has any lock on D,
    if Ti has a lock-S on D then
        upgrade lock on D to lock-X
      else
        grant Ti a lock-X on D
      write (D)
 end;
```

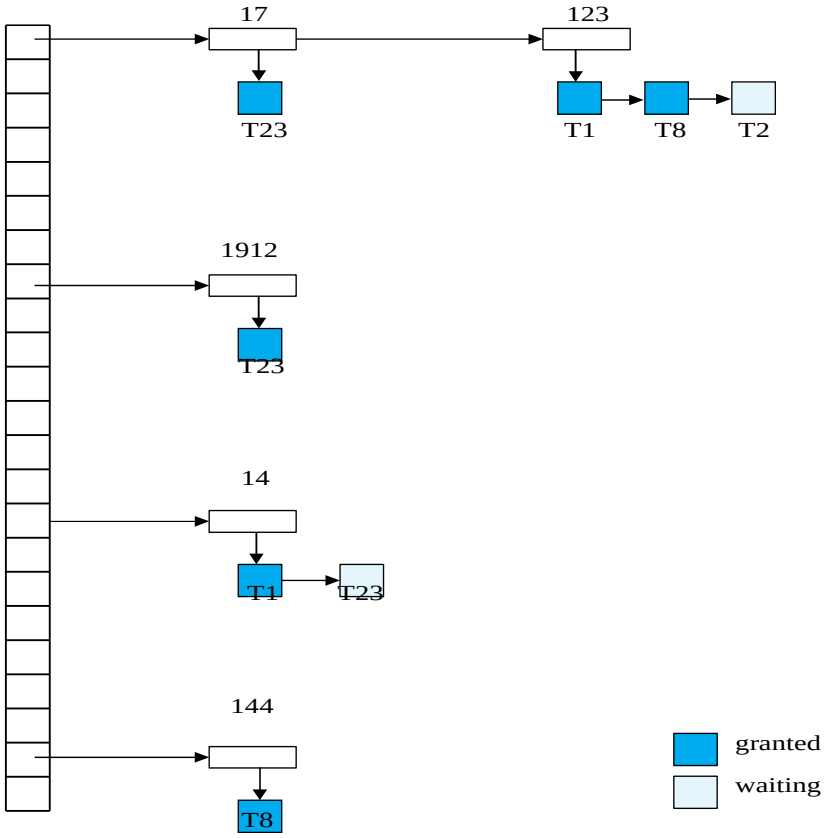
All locks are released after commit or abort

Implementation of Locking:

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests

Lock Table: The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked. Lock table also records the type of lock granted or requested

- Blue rectangles indicate granted locks, Light Blue rectangles ones indicate waiting requests
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted. Lock manager may keep a list of locks held by each transaction, to implement this efficiently



Timestamp-Based Protocols:

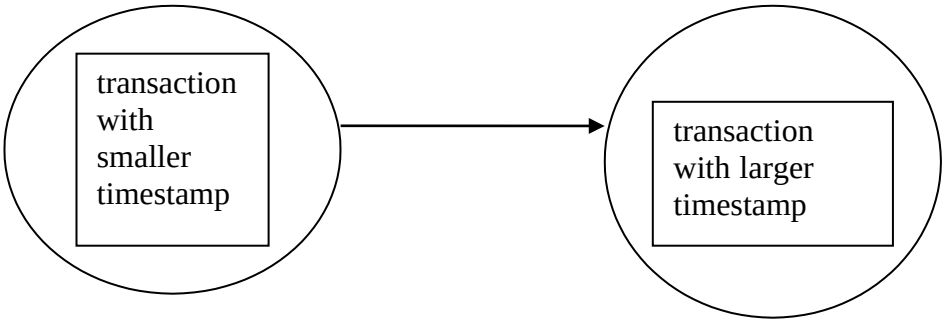
Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS (*Ti*), a new transaction *Tj* is assigned time-stamp TS (*Tj*) such that TS (*Ti*) < TS (*Tj*).

- The protocol manages concurrent execution such that the time-stamps determine the Serializability order.
- In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
  - **W-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **write** (*Q*) successfully.
  - **R-timestamp** (*Q*) is the largest time-stamp of any transaction that executed **read** (*Q*) successfully.
- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction Ti issues a **read**(*Q*)
  - If TS(*Ti*) ≤ **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten.
    - Hence, the **read** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*)≥ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(*Ti*)).
- Suppose that transaction *Ti* issues **write**(*Q*).
  - If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced.
    - Hence, the **write** operation is rejected, and *Ti* is rolled back.
  - If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.
    - Hence, this **write** operation is rejected, and *Ti* is rolled back.
  - Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

Example Use of the Protocol: A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T1	T2	T3	T4	T5
				read(*X*)
	read(Y)			
read(Y)				
		write(*Y*)		
		write(Z)		
				read(Z)
	read(*X*)			
	abort			
read(*X*)				
		write(Z)		
		abort		
				write(*Y*)
				write(Z)

Correctness of Timestamp-Ordering Protocol: The timestamp-ordering protocol guarantees Serializability since all the arcs in the precedence graph are of the form:



- ✓ Thus, there will be no cycles in the precedence graph

Timestamp protocol ensures freedom from deadlock as no transaction ever waits. But the schedule may not be cascade-free, and may not even be recoverable.

Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose *Ti* aborts, but *Tj* has read a data item written by *Ti*
  - Then *Tj* must abort; if *Tj* had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by *Tj* must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks
- Solution 1:
  - A transaction is structured such that its writes are all performed at the end of its processing
  - All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
  - A transaction that aborts is restarted with a new timestamp
- Solution 2: Limited form of locking: wait for data to be committed before reading it
- Solution 3: Use commit dependencies to ensure recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When *Ti* attempts to write data item *Q*, if TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of {*Q*}.
    - ***Rather than rolling back Ti as the timestamp ordering protocol would have done, this {write} operation can be ignored***.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency.
    - Allows some view-serializable schedules that are not conflict-serializable.

Validation-Based Protocol

Execution of transaction *Ti* is done in three phases.

1) Read and execution phase: Transaction *Ti* writes only to temporary local variables
2) Validation phase: Transaction *Ti* performs a "validation test'' to determine if local variables can be written without violating Serializability.
3) Write phase: If *Ti* is validated, the updates are applied to database; otherwise, Ti is rolled back.

The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

Assume for simplicity that the validation and write phase occur together, atomically and serially i.e., only one transaction executes validation/write at a time. Also called as optimistic concurrency control since transaction executes fully in the hope that all will go well during validation

Each transaction Ti has 3 timestamps

1. Start(Ti) : the time when Ti started its execution
2. Validation(Ti): the time when Ti entered its validation phase
3. Finish(Ti) : the time when Ti finished its write phase

Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus TS(Ti) is given the value of Validation(Ti).

This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.

- because the Serializability order is not pre-decided, and
- Relatively few transactions will have to be rolled back.

Validation Test for Transaction *Tj*

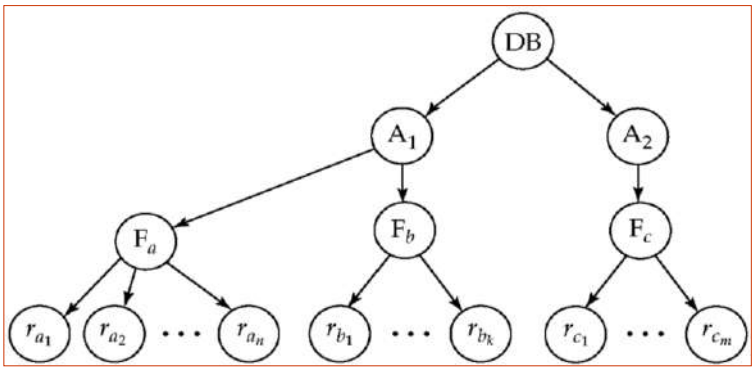
- If for all *Ti* with TS (*Ti*) < TS (*Tj*) either one of the following condition holds:
    - **finish**(*Ti*) < **start**(*Tj*)
    - **start**(*Tj*) < **finish**(*Ti*) < **validation**(*Tj*) **and** the set of data items written by *Ti* does not intersect with the set of data items read by *Tj*.
- Then validation succeeds and *Tj* can be committed. Otherwise, validation fails and *Tj* is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
    - Writes of *Tj* do not affect reads of *Ti* since they occur after *Ti* has finished its reads.
    - Writes of *Ti* do not affect reads of *Tj* since *Tj* does not read any item written by *Ti.*

Schedule Produced by Validation: Example of schedule produced using validation

T14	T15
read(*B*)	
	read(*B*)
	B:= *B-50*
	read(*A*)
	A:= *A+50*
read(*A*)	
(*validate*)	
display (*A*+*B*)	
	(*validate*)
	write (*B*)
	write (*A*)

MULTIPLE GRANULARITY

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all nodes descendents in the same mode.
- Granularity of locking (level in tree where locking is done):
  - **fine granularity** (lower in tree): high concurrency, high locking overhead
  - **coarse granularity** (higher in tree): low locking overhead, low concurrency



The levels, starting from the coarsest (top) level are

1. *database*
2. *area*
3. *file*
4. *record*

Intention Lock Modes: In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

1. ***intention-shared*** (IS): Indicates explicit locking at a lower level of the tree but only with shared locks.
2. ***intention-exclusive*** (IX): Indicates explicit locking at a lower level with exclusive or shared locks
3. ***shared and intention-exclusive*** (SIX): The sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

✓ Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes: The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme: Transaction *Ti* can lock a node *Q*, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node *Q* can be locked by *Ti* in S or IS mode only if the parent of *Q* is currently locked by *Ti* in either IX or IS mode.
4. A node *Q* can be locked by *Ti* in X, SIX, or IX mode only if the parent of *Q* is currently locked by *Ti* in either IX or SIX modes.
5. *Ti* can lock a node only if it has not previously unlocked any node (that is, *Ti* is two-phase).
6. *Ti* can unlock a node *Q* only if none of the children of *Q* are currently locked by *Ti.*

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

MULTIVERSION SCHEMES

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful **write** results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

Multiversion Timestamp Ordering:

- Each data item *Q* has a sequence of versions <*Q1, Q2,... , Qm*>. Each version *Qk* contains three data fields:
  - **Content** -- the value of version *Qk.*
  - **W-timestamp**(*Qk*) -- timestamp of the transaction that created (wrote) version Qk
  - **R-timestamp**(*Qk*) -- largest timestamp of a transaction that successfully read version Qk
- When a transaction *Ti* creates a new version *Qk* of *Q*, Qk's W-timestamp and R-timestamp are initialized to TS(*Ti*).
- R-timestamp of *Qk* is updated whenever a transaction *Tj* reads *Qk*, and TS(*Tj*) > R-timestamp(*Qk*).
- Suppose that transaction *Ti* issues a **read**(*Q*) or **write**(*Q*) operation. Let *Qk* denote the version of *Q* whose write timestamp is the largest write timestamp less than or equal to TS(*T*i).
  - If transaction *Ti* issues a **read**(*Q*), then the value returned is the content of version Qk.
  - If transaction *Ti* issues a **write**(*Q*)
    - if TS(*Ti*) < R-timestamp(*Qk*), then transaction *Ti* is rolled back.
    - if TS(*T*i) = W-timestamp(*Qk*), the contents of *Qk* are overwritten
    - else a new version of *Q* is created.

- Observe that
  - Reads always succeed
  - A write by *Ti* is rejected if some other transaction *Tj* that (in the serialization order defined by the timestamp values) should read *Ti*'s write, has already read a version created by a transaction older than *Ti*.
- ✓ *This Protocol guarantees Serializability.*

Multiversion Two-Phase Locking:

- Differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - Each version of a data item has a single timestamp whose value is obtained from a counter **ts-counter** that is incremented during commit processing.
- *Read-only transactions* are assigned a timestamp by reading current value of **ts-counter** before they start execution; they follow Multiversion timestamp-ordering protocol to perform reads.
- When an update transaction wants to read a data item:
  - It obtains a shared lock on it, and reads the latest version.
- When it wants to write an item
  - It obtains X lock on; it then creates a new version of the item and sets this version's timestamp to ∞.
- When update transaction *Ti* completes, commit processing occurs:
  - *Ti* sets timestamp on the versions it has created to **ts-counter** + 1
  - *Ti* increments **ts-counter** by 1
- Read-only transactions that start after *Ti* increments **ts-counter** will see values updated by *Ti*.
- Read-only transactions that start before *Ti* increments the **ts-counter** will see the value before the updates by *Ti.*
- *Only serializable schedules are produced.*

Implementation Issues

- Creation of multiple versions increases storage overhead
  - Extra tuples
  - Extra space in each tuple for storing version information
- Versions can, however, be garbage collected
  - Ex: if Q has two versions Q5 and Q9, and the oldest active transaction has timestamp > Q9, then Q5 will never be required again

Deadlock Handling

Consider the following two transactions:

*T*1:	write (*X*)	*T*2:	write(*Y*)
	write(*Y*)		write(*X*)

Schedule with deadlock is given as:

T1	T2
lock-X on *X*	
write (*X*)	
	lock-X on *Y*
	write (*X*)
	wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set to unlock the data item.

Deadlock Prevention: Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

- Require that each transaction locks all its data items before it begins execution (pre declaration).
- Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

Deadlock Prevention Strategies: Following schemes use transaction timestamps for the sake of deadlock prevention alone.

1. **Wait-die scheme** (non-preemptive): Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item
2. **Wound-wait scheme** (preemptive) : Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than *wait-die* scheme.

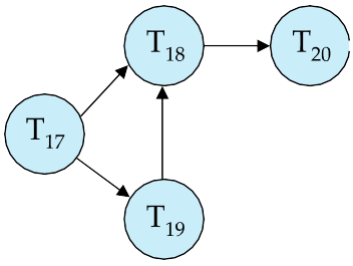
Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

3. **Timeout-Based Schemes:** A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. Thus deadlocks are not possible. It is simple to implement; but starvation is possible. It is also difficult to determine good value of the timeout interval.

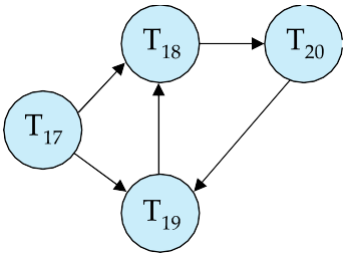
Deadlock Detection: Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$.

- V is a set of vertices (all the transactions in the system)
- E is a set of edges; each element is an ordered pair $Ti \rightarrow Tj$.

- If $Ti \rightarrow Tj$ is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
- When Ti requests a data item currently being held by Tj, then the edge Ti Tj is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery: When a deadlock is detected:

- Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
- Rollback -- determine how far to roll back transaction
  - Total rollback: Abort the transaction and then restart it.
  - More effective to roll back transaction only as far as necessary to break deadlock.
- Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Insert and Delete Operations:

- If two-phase locking is used :
  - A delete operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into database is given an X-mode lock on the tuple
- Insertions and deletions can lead to the **phantom phenomenon**.
  - A transaction that scans a relation (Ex: find sum of balances of all accounts in Perryridge) and a transaction that inserts a tuple in the relation (Ex: insert a new account at Perryridge) conflict (conceptually) in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - Ex: the scan transaction does not see the new account, but reads some other tuple written by the update transaction
- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information. The information should be locked.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.
- Above protocol provides very low concurrency for insertions/deletions.
- Index locking protocols provide higher concurrency while preventing the phantom phenomenon, by requiring locks on certain index buckets.

Index Locking Protocol: According to this protocol every relation must have at least one index. A transaction can access tuples only after finding them through one or more indices on the relation

- A transaction *Ti* that performs a lookup must lock all index leaf nodes that it accesses, in S-mode
  - Even if the leaf node does not contain any tuple satisfying the index lookup
- A transaction *Ti* that inserts, updates or deletes a tuple *ti* in a relation *r*
  - must update all indices to *r*
  - must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
- The rules of the two-phase locking protocol must be observed
- Guarantees that phantom phenomenon won't occur

Concurrency in Index Structures: Indices are unlike other database items in that their only job is to help in accessing data.

- ✓ Index-structures are typically accessed very often, much more than other database items.
  - Treating index-structures like other database items, Ex: by 2-phase locking of index nodes can lead to low concurrency.
- ✓ There are several index concurrency protocols where locks on internal nodes are released early, and not in a two-phase fashion.

- It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.
  - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

✓ Example of index concurrency protocol:

✓ Use **crabbing** instead of two-phase locking on the nodes of the B^+-tree, as follows. During search/insertion/deletion:
- First lock the root node in shared mode.
- After locking all required children of a node in shared mode, release the lock on the node.
- During insertion/deletion, upgrade leaf node locks to exclusive mode.
- When splitting or coalescing requires changes to a parent, lock parent in exclusive mode.

✓ Above protocol can cause excessive deadlocks
- Searches coming down the tree deadlock with updates going up the tree
- Can abort and restart search, without affecting transaction

✓ Better protocols are available; such as the B-link tree protocol
- Intuition: Release lock on parent before acquiring lock on child and deal with changes that may have happened between lock release and acquire

Weak Levels of Consistency

➢ **Degree-two consistency:** differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
- X-locks must be held till end of transaction
- Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur.

➢ **Cursor stability**:
- For reads, each tuple is locked, read, and lock is immediately released
- X-locks are held till end of transaction
- Special case of degree-two consistency

Weak Levels of Consistency in SQL: SQL allows non-serializable executions

✓ **Serializable:** is the default

✓ **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
- However, the phantom phenomenon need not be prevented
  - T1 may see some records inserted by T2, but may not see others inserted by T2

✓ **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability

✓ **Read uncommitted**: allows even uncommitted data to be read

In many database systems, read committed is the default consistency level and has to be explicitly changed to serializable when required as:

n **set isolation level serializable**

RECOVERY SYSTEM

Failure Classification: Failures are basically classified into three. They are:

1. **Transaction Failure**:
   - ✓ **Logical errors**: Transaction cannot complete due to some internal error condition
   - ✓ **System errors**: The database system must terminate an active transaction due to an error condition (Ex: deadlock)
2. **System Crash**: A power failure or other hardware or software failure causes the system to crash.
   - ✓ **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
   - ✓ Database systems have numerous integrity checks to prevent corruption of disk data
3. **Disk Failure**: A head crash or similar disk failure destroys all or part of disk storage
   - ✓ Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms: Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures. Recovery algorithms have two parts:

- Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

Storage Structure:

1. **Volatile storage**: Does not survive system crashes. Examples: main memory, cache memory
2. **Nonvolatile storage**: Survives system crashes. Examples: disk, tape, flash memory
3. **Stable storage**: A mythical form of storage that survives all failures. Approximated by maintaining multiple copies on distinct nonvolatile media.

Stable-Storage Implementation

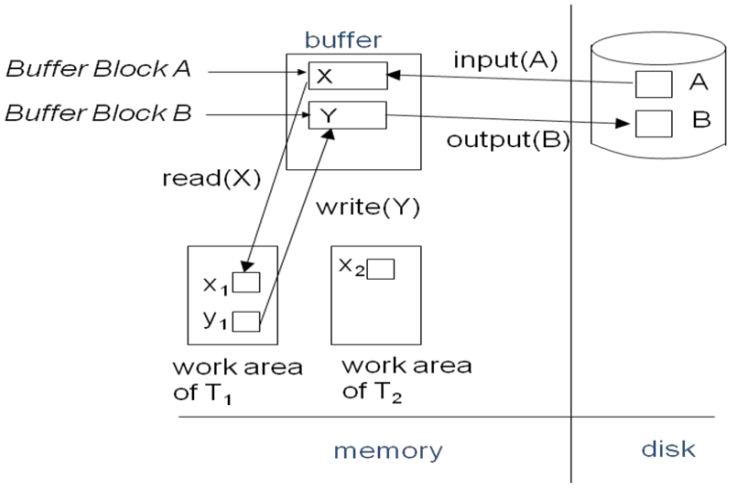
- Maintain multiple copies of each block on separate disks
  - Copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies: Block transfer can result in
  - Successful completion
  - ***Partial Failure***: destination block has incorrect information
  - ***Total Failure***: destination block was never updated
- Protecting storage media from failure during data transfer:
  - Execute output operation as follows (assuming two copies of each block):
    - Write the information onto the first physical block.
    - When the first write successfully completes, write the same information onto the second physical block.
    - The output is completed only after the second write successfully completes.
- Copies of a block may differ due to failure during output operation. To recover from failure:
  - First find inconsistent blocks:
    - **Expensive solution**: Compare the two copies of every disk block.
    - **Better solution**: Record in-progress disk writes on non-volatile storage. Use this information during recovery to find blocks that may be inconsistent, and only compare copies of these. Used in hardware RAID systems
  - If either copy of an inconsistent block is detected to have an error (bad checksum), overwrite it by the other copy. If both have no error, but are different, overwrite the second block by the first block.

Data Access: In memory, **Physical blocks** are those blocks residing on the disk. **Buffer blocks** are the blocks residing temporarily in main memory.

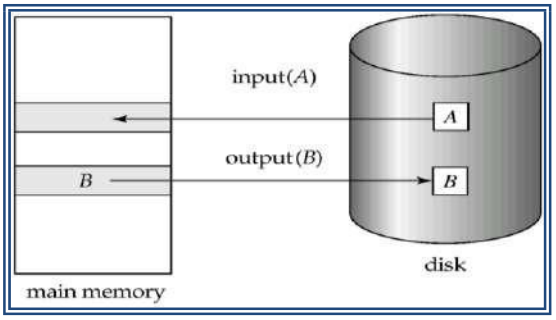
- Block movements between disk and main memory are initiated through the following two operations:
  - **input** (*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- Each transaction *Ti* has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - *Ti*'s local copy of a data item *X* is called *xi.*
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.
- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read**(*X*) assigns the value of data item *X* to the local variable *xi*.
  - **write**(*X*) assigns the value of local variable *xi* to data item {*X*} in the buffer block.
  - both these commands may necessitate the issue of an **input**(BX) instruction before the assignment, if the block *BX* in which *X* resides is not already in memory.
- Transactions
  - Perform **read**(*X*) while accessing *X* for the first time;
  - All subsequent accesses are to the local copy.
  - After last access, transaction executes **write**(*X*).
- **output**(*BX*) need not immediately follow **write**(*X*). System can perform the **output** operation when it deems fit.

A buffer block is eventually written out to the disk either because the buffer manager needs the memory space for other purposes or because the database system wishes to reflect the change to *B* on the disk. We shall say that the database system performs a **force-output** of buffer *B* if it issues an output(*B*).

EXAMPLE OF DATA ACCESS



Block Storage Operations:



Recovery and Atomicity:

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction *Ti* that transfers $50 from account *A* to account *B*; goal is either to perform all database modifications made by *Ti* or none at all.
- Several output operations may be required for *Ti* (to output *A* and *B*). A failure may occur after one of these modifications has been made but before all of them are made.
- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.

Log-Based Recovery:

- Log is a sequence of **log records**, and maintains a record of update activities on the database. A log is kept on stable storage.
- When transaction *Ti* starts, it registers itself by writing a <*Ti* **start**>log record
- *Before Ti* executes **write**(*X*), a log record <*Ti, X, V1, V2*> is written, where *V1* is the value of *X* before the write, and *V2* is the value to be written to *X*.
    - Log record notes that *Ti* has performed a write on data item *Xj* *Xj* had value *V1* before the write, and will have value *V2* after the write.
- When *Ti* finishes it last statement, the log record <*Ti* **commi**t> is written.
- We assume for now that log records are written directly to stable storage.
- Two approaches using logs
    - Deferred database modification
    - Immediate database modification

Deferred Database Modification:

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <*Ti* ***start***> record to log.
- A **write**(*X*) operation results in a log record <*Ti, X, V*> being written, where *V* is the new value for *X*
    - Note: old value is not needed for this scheme
- The write is not performed on *X* at this time, but is deferred.
- When *Ti* partially commits, <*Ti* **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Below we show the log as it appears at three instances of time.

$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>
(a)	(b)	(c)

If log on stable storage at time of crash is as in case:

(a) No redo actions need to be taken

(b) redo(*T*0) must be performed since <*T*0 **commi**t> is present

(c) **redo**(*T*0) must be performed followed by redo(*T*1) since <*T*0 **commit**> and <*Ti* commit> are present

Immediate Database Modification: The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued

- since undoing may be needed, update logs must have both old value and new value

- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	**Write**	**Output**
<*T*0 **start**>		
<*T0,* A, 1000, 950>		
To, B, 2000, 2050		
	A = 950	
	B = 2050	
<*T*0 **commit**>		
<*T*1 **start**>		
<*T*1, C, 700, 600>		
	C = 600	
		BB, *BC*
<*T*1 **commit**>		
		BA

Note: *BX* denotes block containing *X*.

- Recovery procedure has two operations instead of one:
  - **undo**(*T*i) restores the value of all data items updated by *Ti* to their old values, going backwards from the last log record for *Ti*
  - **redo**(*T*i) sets the value of all data items updated by *Ti* to the new values, going forward from the first log record for *Ti*
- Both operations must be **idempotent.** i.e., even if the operation is executed multiple times the effect is the same as if it is executed once
  - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction *Ti* needs to be undone if the log contains the record <*Ti* **start**>, but does not contain the record <*Ti* **commit**>.
  - Transaction *Ti* needs to be redone if the log contains both the record <*Ti* **start**> and the record <*Ti* **commit**>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) Undo (*T*0): B is restored to 2000 and A to 1000.

(b) Undo (*T*1) and redo (*T*0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) Redo (*T*0) and redo (*T*1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpoints:

Problems in recovery procedure include:

1. Searching the entire log is time-consuming
2. We might unnecessarily redo transactions which have already
3. Output their updates to the database.

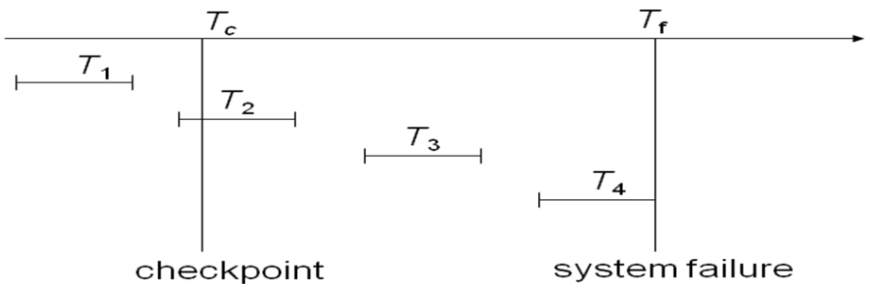
Streamline recovery procedure by periodically performing **checkpointing**

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record <**checkpoint**> onto stable storage.

During recovery we need to consider only the most recent transaction Ti that started before the checkpoint and transactions that started after *Ti*.

1. Scan backwards from end of log to find the most recent <**checkpoint**> record
2. Continue scanning backwards till a record <*Ti* **start**> is found.
3. Need to only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
   - For all transactions (starting from *Ti* or later) with no <*Ti* **commit**>, execute **undo*(Ti).***
   - Scanning forward in the log, for all transactions starting from *Ti* or later with a <*Ti* **commit**>, execute **redo*(Ti).***

Example of Checkpoints



*T*1 can be ignored (updates already output to disk due to checkpoint)

*T*2 and *T*3 redone.

*T*4 undone

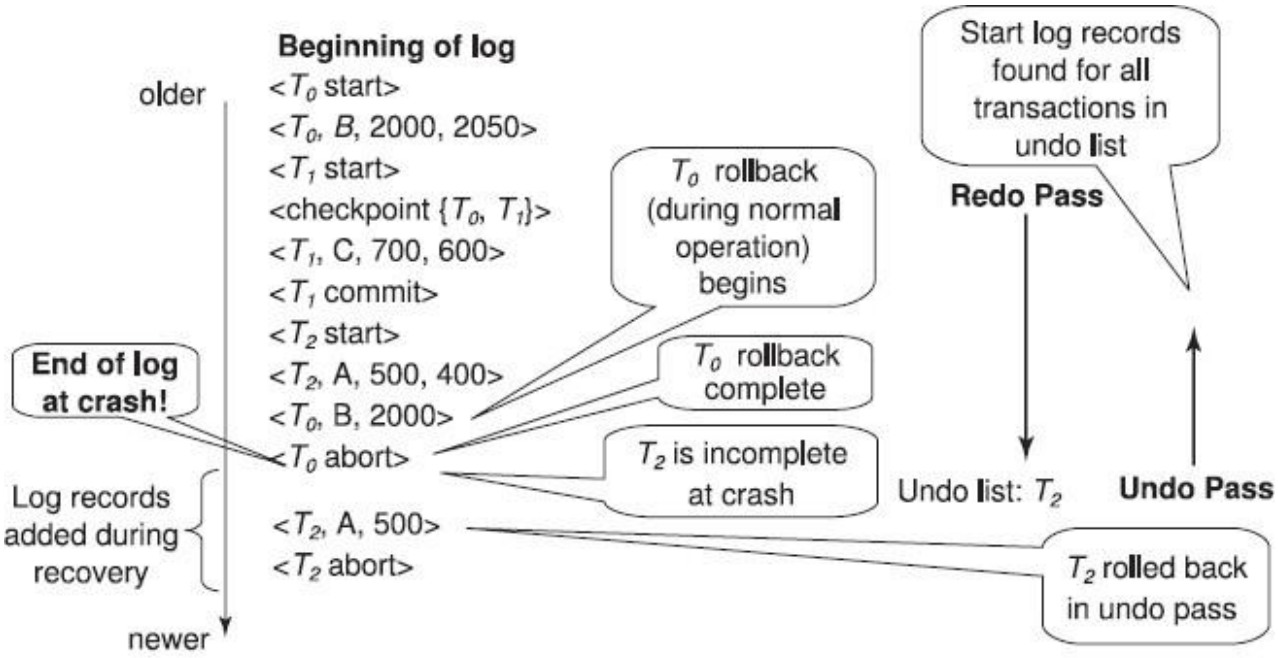
Recovery with Concurrent Transactions: We modify the log-based recovery schemes to allow multiple transactions to execute concurrently. All transactions share a single disk buffer and a single log. A buffer block can have data items updated by one or more transactions

- We assume concurrency control using strict two-phase locking; i.e. the updates of uncommitted transactions should not be visible to other transactions
- Logging is done as described earlier. Log records of different transactions may be interspersed in the log.
- The checkpointing technique and actions taken on recovery have to be changed, since several transactions may be active when a checkpoint is performed.
- The checkpoints are performed as before, except that the checkpoint log record is now of the form < **checkpoint** *L*>, where *L* is the list of transactions active at the time of the checkpoint
    - We assume no updates are in progress while the checkpoint is carried out
- When the system recovers from a crash, it first does the following:
    - Initialize *undo-list* and *redo-list* to empty
    - Scan the log backwards from the end, stopping when the first <**checkpoint** *L*> record is found.

      For each record found during the backward scan:
        - if the record is <*Ti* **commit**>, add *Ti* to *redo-list*
        - if the record is <*Ti* **start**>, then if *Ti* is not in *redo-list*, add *Ti* to *undo-list*
    - For every *Ti* in *L*, if *T*i is not in *redo-list*, add *Ti* to *undo-list*
- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
    - Scan log backwards from most recent record, stopping when <*Ti* **start**> records have been encountered for every *Ti* in *undo-list*.
        - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.
    - Locate the most recent <**checkpoint** *L*> record.
    - Scan log forwards from the <**checkpoint** *L*> record till the end of the log.
        - During the scan, perform **redo** for each log record that belongs to a transaction on *redo-list*

Example of Recovery: Go over the steps of the recovery algorithm on the following log:

<*T*0 **star**t>
<*T*0, *A*, 0, 10>
<*T*0 **commit**>
<*T*1 **start**> /* Scan at step 1 comes up to here */
<*T*1, *B*, 0, 10>
<*T*2 **start**>
<*T*2, *C*, 0, 10>
<*T*2, *C*, 10, 20>
<checkpoint {*T*1, *T*2}>
<*T*3 **start**>
<*T*3, *A*, 10, 20>
<*T*3, *D*, 0, 10>
<*T*3 **commit**>



Example of logged actions, and actions during recovery.

Buffer Management:

Log Record Buffering: Log records are buffered in main memory, instead of of being output directly to stable storage. Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.

- ✓ Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- ✓ Several log records can thus be output using a single output operation, reducing the I/O cost.
- ✓ The rules below must be followed if log records are buffered:
    - Log records are output to stable storage in the order in which they are created.
    - Transaction *Ti* enters the commit state only when the log record <*Ti* **commit**> has been output to stable storage.
    - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage. This rule is called the **write-ahead logging** or **WAL** rule
        - Strictly speaking WAL only requires undo information to be output

Database Buffering: Database maintains an in-memory buffer of data blocks. When a new block is needed, if buffer is full an existing block needs to be removed from buffer. If the block chosen for removal has been updated, it must be output to disk

- ➢ If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
- ➢ No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
    - Before writing a data item, transaction acquires exclusive lock on block containing the data item
    - Lock can be released once the write is completed.
        - Such locks held for short duration are called **latches**.
    - Before a block is output to disk, the system acquires an exclusive latch on the block
        - Ensures no update can be in progress on the block

Buffer Management:

- ✓ Database buffer can be implemented either in an area of real main-memory reserved for the database, or in virtual memory
- ✓ Implementing buffer in reserved main-memory has *drawbacks*:
  1. Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  2. Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.
- ✓ Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - ➢ When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - ➢ When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!. This is known as **dual paging** problem.
  - ➢ Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    - Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    - Release the page from the buffer, for the OS to use

    Dual paging can thus be avoided, but common operating systems do not support such functionality.

Failure with Loss of Nonvolatile Storage: A technique similar to checkpointing is used to deal with loss of non-volatile storage

- ✓ Periodically **dump** the entire content of the database to stable storage
- ✓ No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
  - ❖ Output all log records currently residing in main memory onto stable storage.
  - ❖ Output all buffer blocks onto the disk.
  - ❖ Copy the contents of the database to stable storage.
  - ❖ Output a record <**dump**> to log on stable storage.

Recovering from Failure of Non-Volatile Storage: To recover from disk failure, Restore DB from most recent dump. Consult log and redo all transactions that committed after the dump.

- ➢ Can be extended to allow transactions to be active during dump; known as **fuzzy dump** or **online dump**

Advanced Recovery Algorithm

Key Features:

- Support for high-concurrency locking techniques, which release locks early.
- Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing.
  - including redo of log records of incomplete transactions, followed by subsequent undo
- Supports logical undo.
- Easier to understand/show correctness.

Logical Undo Logging: Operations like B^+ tree insertions and deletions release locks early. They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+ tree. Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).

- ✓ For such operations, undo log records should contain the undo operation to be executed. Such logging is called **logical undo logging**, in contrast to **physical undo logging**. Operations are called **logical operations.**
- ✓ Other examples:
  - delete of tuple, to undo insert of tuple, allows early lock release on space allocation information
  - subtract amount deposited, to undo deposit, allows early lock release on bank balance

Physical Redo: Redo information is logged **physically** (i.e., new value for each write) even for operations with logical undo. Physical redo logging does not conflict with early lock release.
Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts.

Operation Logging: Operation logging is done as follows:

1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
2. While operation is executing, normal log records with physical redo and physical undo information are logged.
3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.

Example: insert of (key, record-id) pair (K5, RID7) into index I9

- ✓ If crash/rollback occurs before operation completes: The **operation-end** log record is not found, and the physical undo information is used to undo operation.
- ✓ If crash/rollback occurs after the operation completes:
  - ❖ the **operation-end** log record is found, and in this case
  - ❖ logical undo is performed using U; physical undo information for the operation is ignored.
- ✓ Redo of operation (after crash) still uses physical redo information.

Transaction Rollback: Rollback of transaction T_i is done as follows:
Scan the log backwards,

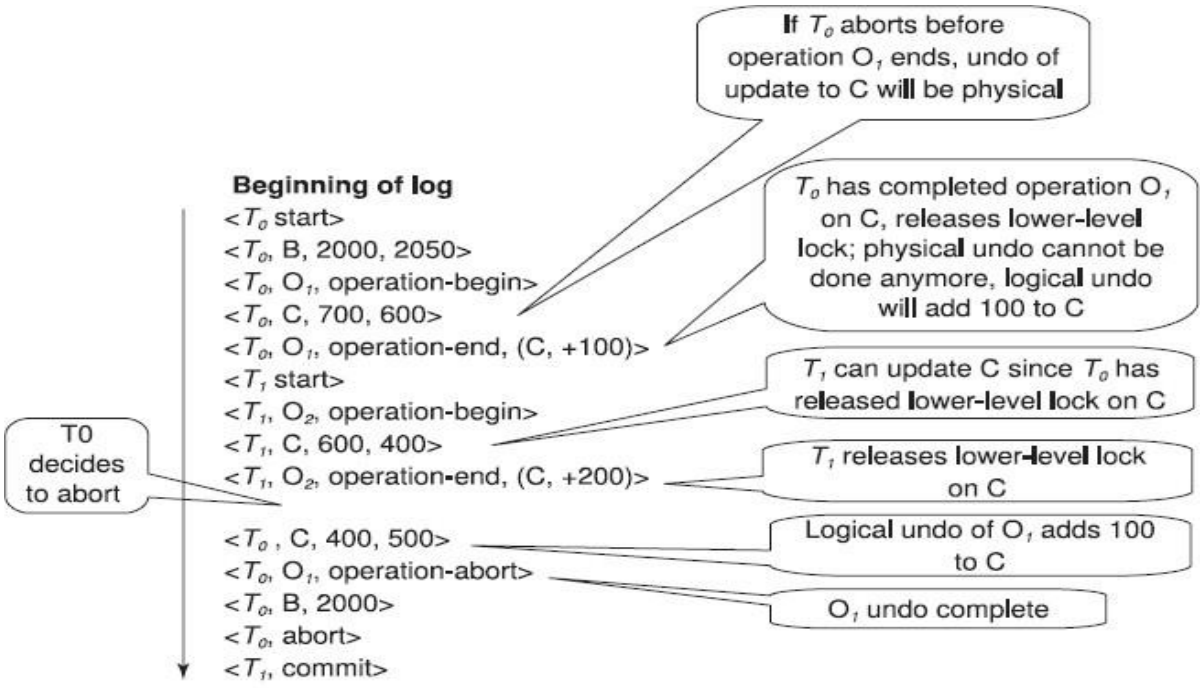
1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a special **redo-only log record** <T_i, X, V_1>.
2. If a <T_i, O_j, **operation-end**, U> record is found
   - ✓ Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generate a record <T_i, O_j, **operation-abort**>.
   - ✓ Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found
3. If a redo-only record is found ignore it
4. If a <T_i, O_j, **operation-abort**> record is found:
   - ✓ skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
5. Stop the scan when the record <T_i, start> is found
6. Add a <T_i, **abort**> record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Transaction Rollback Example: Example with a complete and an incomplete operation

```
<T1, start>
<T1, O1, operation-begin>
….
<T1, X, 10, K5>
<T1, Y, 45, RID7>
<T1, O1, operation-end, (delete I9, K5, RID7)>
<T1, O2, operation-begin>
<T1, Z, 45, 70>
                    ← T1 Rollback begins here
<T1, Z, 45>      ← redo-only log record during physical undo (of incomplete O2)
<T1, Y, .., ..>   ← Normal redo records for logical undo of O1
    …
<T1, O1, operation-abort> ← What if crash occurred immediately after this?
<T1, abort>
```



Transaction rollback with logical undo operations.

Crash Recovery: The following actions are taken when recovering from system crash

1. **Redo Phase**: Scan log forward from last < **checkpoint** *L*> record till end of log
    1. **Repeat history** by physically redoing all updates of all transactions,
    2. Create an undo-list during the scan as follows
        - *undo-list* is set to *L* initially
        - Whenever <T_i **start**> is found T_i is added to *undo-list*
        - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from *undo-list*

This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.

Now *undo-list* contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.

2. **Undo Phase**: Scan log backwards, performing undo on log records of transactions found in *undo-list*.
   - Log records of transactions being rolled back are processed as described earlier, as they are found
     - Single shared scan for all transactions being undone
   - When <T_i **start**> is found for a transaction T_i in *undo-list*, write a <T_i **abort**> log record.
   - Stop scan when <T_i **start**> records have been found for all T_i in *undo-list*

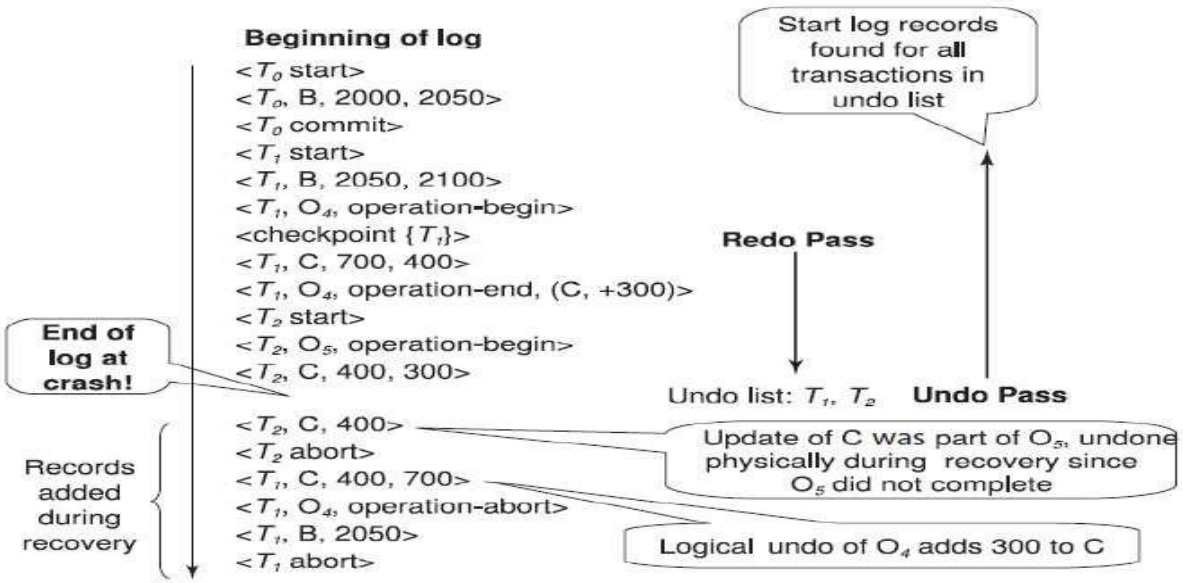
This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Checkpointing: **Checkpointing** is done as follows:

1. Output all log records in memory to stable storage
2. Output to disk all modified buffer blocks
3. Output to log on stable storage a < **checkpoint** *L*> record.

Transactions are not allowed to perform any actions while checkpointing is in progress.

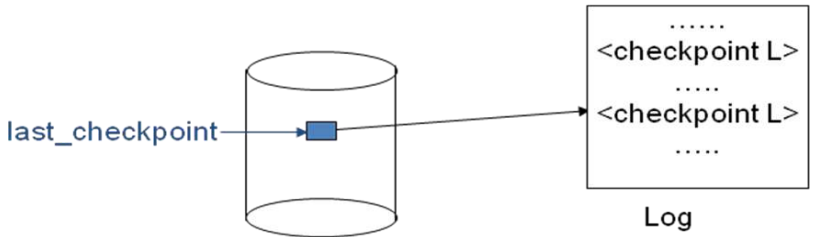
- Fuzzy checkpointing allows transactions to progress while the most time consuming parts of checkpointing are in progress



Failure recovery actions with logical undo operations

Fuzzy Checkpointing: Fuzzy checkpointing is done as follows:

1. Temporarily stop all updates by transactions
2. Write a <**checkpoint** *L*> log record and force log to stable storage
3. Note list *M* of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list *M*
   - ✓ blocks should not be updated while being output
   - ✓ Follow WAL: all log records pertaining to a block must be output before the block is output
6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk



- When recovering using a fuzzy checkpoint, start scan from the **checkpoint** record pointed to by **last_checkpoint**
  - Log records before **last_checkpoint** have their updates reflected in database on disk, and need not be redone.
  - Incomplete checkpoints, where system had crashed while performing checkpoint, are handled safely

ARIES Recovery Algorithm: ARIES is a state of the art recovery method. It incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery. The "advanced recovery algorithm" we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations

- ✓ Unlike the advanced recovery algorithm, ARIES
  - Uses **log sequence number (LSN)** to identify log records
    - Stores LSNs in pages to identify what updates have already been applied to a database page
  - Physiological redo
  - Dirty page table to avoid unnecessary redos during recovery
  - Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time

ARIES Optimizations:

Physiological redo: Affected page is physically identified, action within page can be logical

- ❖ Used to reduce logging overheads
  - Ex: when a record is deleted and all other records have to be moved to fill hole
    - Physiological redo can log just the record deletion
    - Physical redo would require logging of old and new values for much of the page
- ❖ Requires page to be output to disk atomically
  - Easy to achieve with hardware RAID, also supported by some disk systems
  - Incomplete page output can be detected by checksum techniques,
    - But extra actions are required for recovery
    - Treated as a media failure

ARIES Data Structures: ARIES uses several data structures

- ❖ Log sequence number (LSN) identifies each log record
  - Must be sequentially increasing
  - Typically an offset from beginning of log file to allow fast access. Easily extended to handle multiple log files
- ❖ Page LSN
- ❖ Log records of several different types
- ❖ Dirty page table

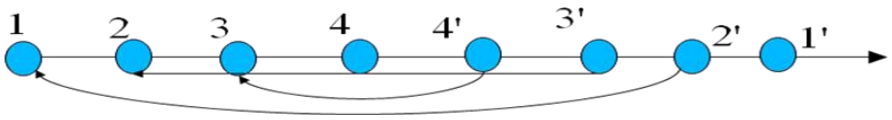
1) **Page LSN:** Each page contains a **PageLSN** which is the LSN of the last log record whose effects are reflected on the page
   - ✓ To update a page:
     - X-latch the page, and write the log record
     - Update the page
     - Record the LSN of the log record in PageLSN
     - Unlock page
   - ✓ To flush page to disk, must first S-latch page
     - Thus page state on disk is operation consistent.
       - Required to support physiological redo
   - ✓ PageLSN is used during recovery to prevent repeated redo, thus ensuring idempotence
2) **Log Record:** Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo

   - LSN in log record may be implicit
   - ✓ Special redo-only log record called **compensation log record (CLR)** used to log actions taken during recovery that never need to be undone
     - Serves the role of operation-abort log records used in advanced recovery algorithm
     - Has a field UndoNextLSN to note next (earlier) record to be undone

LSN	TransID	UndoNextLSN	RedoInfo

   - ✓ Records in between would have already been undone
   - ✓ Required to avoid repeated undo of already undone actions



3) **DirtyPage Table:** List of pages in the buffer that have been updated contains,
   - **PageLSN** of the page
   - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
     - Set to current end of log when a page is inserted into dirty page table
     - Recorded in checkpoints, helps to minimize redo work

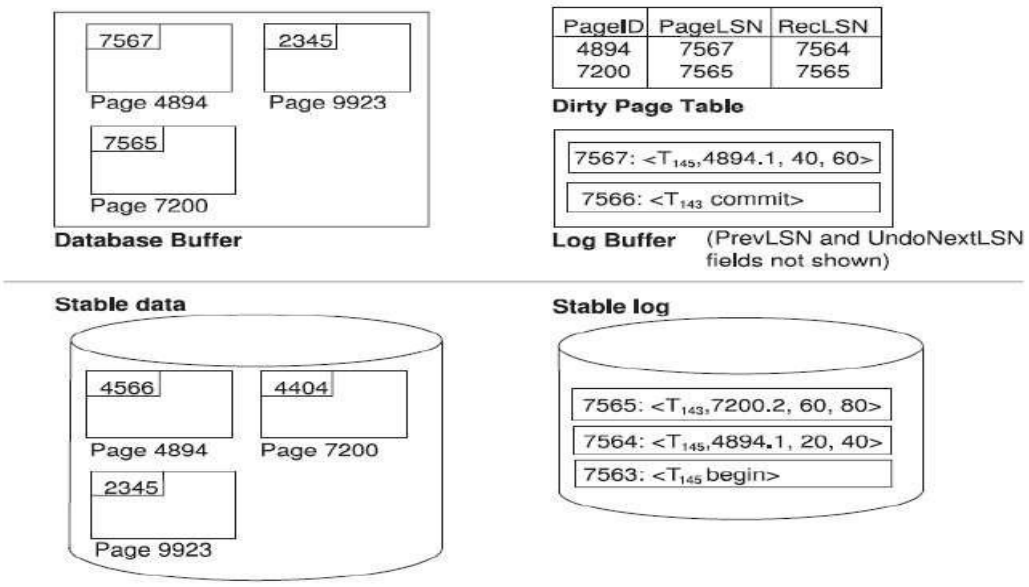


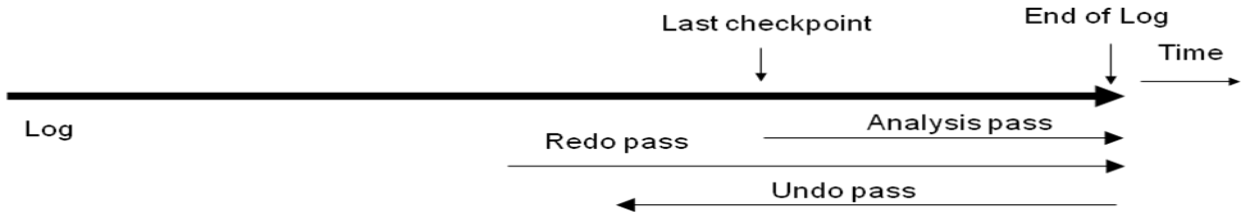
Figure 16.8 Data structures used in ARIES.

4) Checkpoint Log: Contains:

- DirtyPageTable and list of active transactions
- For each active transaction, LastLSN, the LSN of the last log record written by the transaction
  - Fixed position on disk notes LSN of last completed checkpoint log record
- Dirty pages are not written out at checkpoint time. Instead, they are flushed out continuously, in the background
- Checkpoint is thus very low overhead. It can be done frequently

ARIES Recovery Algorithm: ARIES recovery involves three passes

1. ***Analysis Pass:*** Determines
   - Which transactions to undo
   - Which pages were dirty (disk version not up to date) at time of crash
   - RedoLSN: LSN from which redo should start
2. ***Redo Pass:*** Repeats history, redoing all actions from RedoLSN
   - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
3. ***Undo Pass:*** Rolls back all incomplete transactions. Transactions whose abort was complete earlier are not undone
   - Key idea: no need to undo these transactions: earlier undo actions were logged, and are redone as required



1. **ARIES Analysis Pass:** Analysis determines where redo should start. It starts from last complete checkpoint log record
   - Reads DirtyPageTable from log record
   - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
     - In case no pages are dirty, RedoLSN = checkpoint record's LSN
   - Sets undo-list = list of transactions in checkpoint log record
   - Reads LSN of last log record for each transaction in undo-list from checkpoint log record

❖ Scans forward from checkpoint

- If any log record found for transaction not in undo-list, adds transaction to undo-list
- Whenever an update log record is found
  - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
- If transaction end log record found, delete transaction from undo-list
- Keeps track of last log record for each transaction in undo-list
  - May be needed for later undo

❖ At end of analysis pass:

- RedoLSN determines where to start redo pass
- RecLSN for each page in DirtyPageTable used to minimize redo work
- All transactions in undo-list need to be rolled back

2. **ARIES Redo Pass:** Repeats history by replaying every action not already reflected in the page on disk, as follows:
   - Scans forward from RedoLSN. Whenever an update log record is found:
     1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record

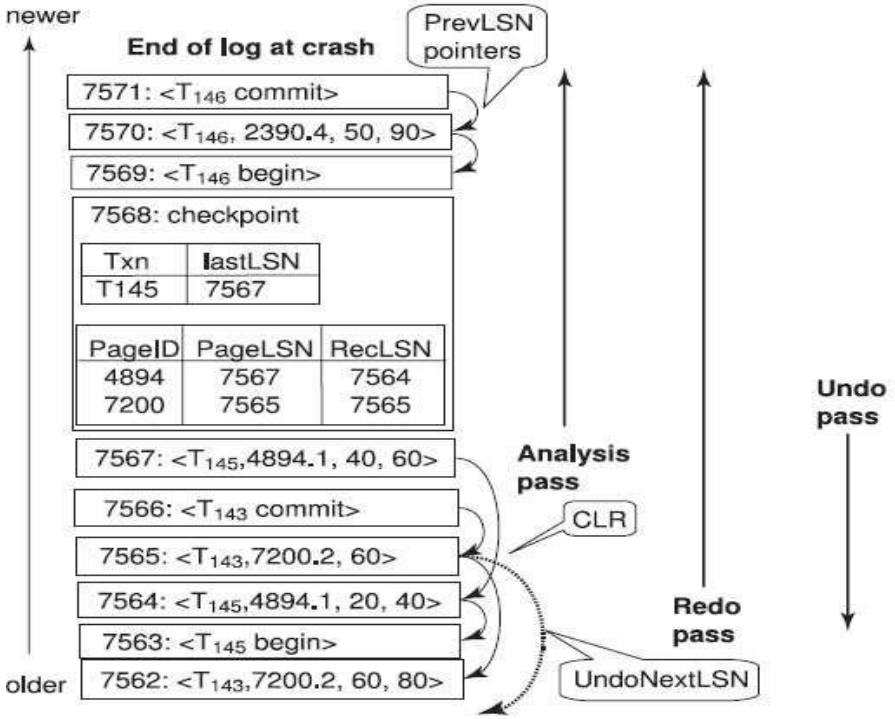
2. Otherwise fetch the page from disk. If the PageLSN of the page fetched from disk is less than the LSN of the log record, redo the log record

NOTE: if either test is negative the effects of the log record have already appeared on the page. First test avoids even fetching the page from disk.

3. **ARIES Undo Pass:** Performs backward scan on log undoing all transaction in undo-list. Undo has to go back till start of earliest incomplete transaction

❖ Backward scan optimized by skipping unneeded log records as follows:

- ✓ Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
- ✓ At each step pick largest of these LSNs to undo, skip back to it and undo it
- ✓ After undoing a log record
  - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record
  - For compensation log records (CLRs) set next LSN to be undo to UndoNextLSN noted in the log record
    - All intervening records are skipped since they would have been undone already
- ✓ Undos are performed as described earlier

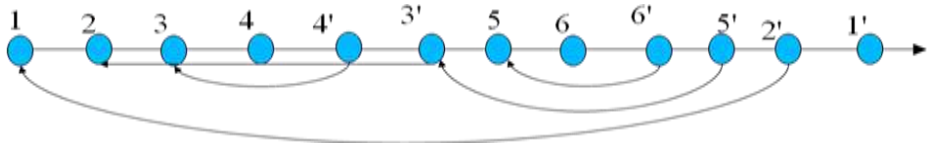


Recovery actions in ARIES.

ARIES Undo Actions: When an undo is performed for an update log record

- Generate a CLR containing the undo action performed (actions performed during undo are logged physicaly or physiologically).
  - CLR for record *n* noted as *n'* in figure below
- Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
  - Arrows indicate UndoNextLSN value

- ARIES supports partial rollback
  - Used Ex: to handle deadlocks by rolling back just enough to release reqd. locks
  - Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback

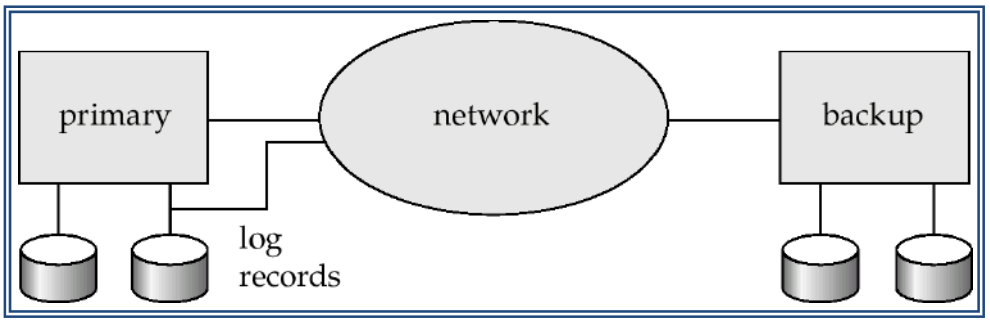


Other ARIES Features:

1. **Recovery Independence**: Pages can be recovered independently of others.
   - Ex: if some disk pages fail they can be recovered from a backup while other pages are being used
2. **Savepoints:** Transactions can record savepoints and roll back to a savepoint.
   - Useful for complex transactions
   - Also used to rollback just enough to release locks on deadlock
3. **Fine-Grained Locking:** Index concurrency algorithms that permit tuple level locking on indices can be used
   - These require logical undo, rather than physical undo, as in advanced recovery algorithm
4. **Recovery optimizations:** Dirty page table can be used to prefetch pages during redo
   - Out of order redo is possible:
     - redo can be postponed on a page being fetched from disk, and performed when page is fetched.
     - Meanwhile other log records can continue to be processed

Remote Backup Systems:

Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.



- ✓ **Detection of failure**: Backup site must detect when primary site has failed. To distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
- ✓ **Transfer of control**: To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary. Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.
- ✓ **Time to recover**: To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- ✓ **Hot-Spare:** Configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.
- ✓ **Alternative to remote backup:** Distributed database with replicated data. Remote backup is faster and cheaper, but less tolerant to failure

Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.

1. **One-safe:** commit as soon as transaction's commit log record is written at primary
    - Problem: updates may not arrive at backup before it takes over.
2. **Two-very-safe:** commit when transaction's commit log record is written at primary and backup
    - Reduces availability since transactions cannot commit if either site fails.
3. **Two-safe:** proceed as in two-very-safe if both primary and backup are active. If only the primary is active, transaction commits as soon as is commit log record is written at primary.
    - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.