ELECTRONICS AND COMPUTER ENGINEERING

LECTURE NOTES

Data Base Management System

J.B. INSTITUTE OF ENGINEERING AND TECHNOLOGY
(UGC AUTONOMOUS)
Bhaskar Nagar, Moinabad Mandal , R.R. District, Hyderabad -500075

UNIT-I

Introduction to Database systems

Basic definitions

DBMS Application

File Systems Vs DBMS

View of DATA

History of Database Systems

Data models

Network and hierarchal model

Relational model

E-R model

Features of the ER model

ER diagram

Conceptual design with ER model

Relational model

Integrity constraints of relations

Enforcing Integrity constraints

Database languages

DB Language (DML, DDL)

DB Users and Administrator

Data storage and Querying

DBMS Architecture

UNIT-1

CONCEPTUAL MODELLING

Introduction to Database systems

DATABASE:-A database is a collection of information that is organized so that it can be easily accessed, managed and updated. Data is organized into rows, columns and tables, and it is indexed to make it easier to find relevant information. Data gets updated, expanded and deleted as new information is added. Databases process workloads to create and update themselves, querying the data they contain and running applications against it. **DATA: -** Any factor that can be stored.

Example: text, numbers, images, videos and speech.

Database Applications: A Database application is a computer program whose primary purpose is entering and retrieving information from a computerized database.

Banking: all transactions

Airlines: reservations, schedules

Universities: registration, grades

Sales: customers, products, purchases

Online retailers: order tracking, customized recommendations

Manufacturing: production, inventory, orders, supply chain

Human resources: employee records, salaries, tax deductions

Databases touch all aspects of our lives

What Is a DBMS?

A *Database Management System (DBMS)* is a software package designed to interact with end-users, other applications, store and manage databases. A general-purpose DBMS allows the definition, creation, querying, update, and administration of databases.

A very large, integrated collection of data.

Models real-world *enterprise.* Entities (e.g., students, courses) Relationships (e.g., Madonna is taking CS564).

DBMS contains information about a particular enterprise

Collection of interrelated data

Set of programs to access the data

An environment that is both *convenient* and *efficient* to use

Why Use a DBMS?

A database management system stores, organizes and manages a large amount of information within a single software application. It manages data efficiently and allows users to perform multiple tasks with ease.

Reduced application development time.

Data integrity and security.

Uniform data administration.

Concurrent access, recovery from crashes.

Why Study Databases??

Shift from *computation* to *information* at the "low end": scramble to webspace (a mess!) at the "high end": scientific applications

Datasets increasing in diversity and volume. Digital libraries, interactive video, Human Genome project, EOS project ... need for DBMS exploding

DBMS encompasses most of CS OS, languages, theory, AI, multimedia, logic.

Purpose of Database Systems:

In the early days, database applications were built directly on top of file systems. A DBMS provides users with a systematic way to create, retrieve, update and manage data. It is a middleware between the databases which store all the data and the users or applications which need to interact with that stored database. A DBMS can limit what data the end user sees, as well as how that end user can view the data, providing many views of a single database schema.

Database + database management system = database system

Drawbacks of using file systems to store data:

Data redundancy and inconsistency.

Multiple file formats, duplication of information in different files.

Difficulty in accessing data.

Need to write a new program to carry out each new task.

Data isolation — multiple files and formats

Integrity problems

Hard to add new constraints or change existing ones

Atomicity of updates

Failures may leave database in an inconsistent state with partial updates carried out

Example: Transfer of funds from one account to another should either complete or not happen at all

Concurrent access by multiple users

Concurrent accessed needed for performance

Uncontrolled concurrent accesses can lead to inconsistencies

Example: Two people reading a balance and updating it at the same time

Security problems

Hard to provide user access to some, but not all, data

Database systems offer solutions to all the above problems

Files vs. DBMS:

A file processing system is a collection of programs that store and manage files in computer hard-disk. On the other hand, a database management system is collection of programs that enables to create and maintain a database. File processing system has more data redundancy, less data redundancy in dbms.

Application must stage large datasets between main memory and secondary storage (e.g., buffering, page-oriented access, 32-bit addressing, etc.)

Special code for different queries

Must protect data from inconsistency due to multiple concurrent users

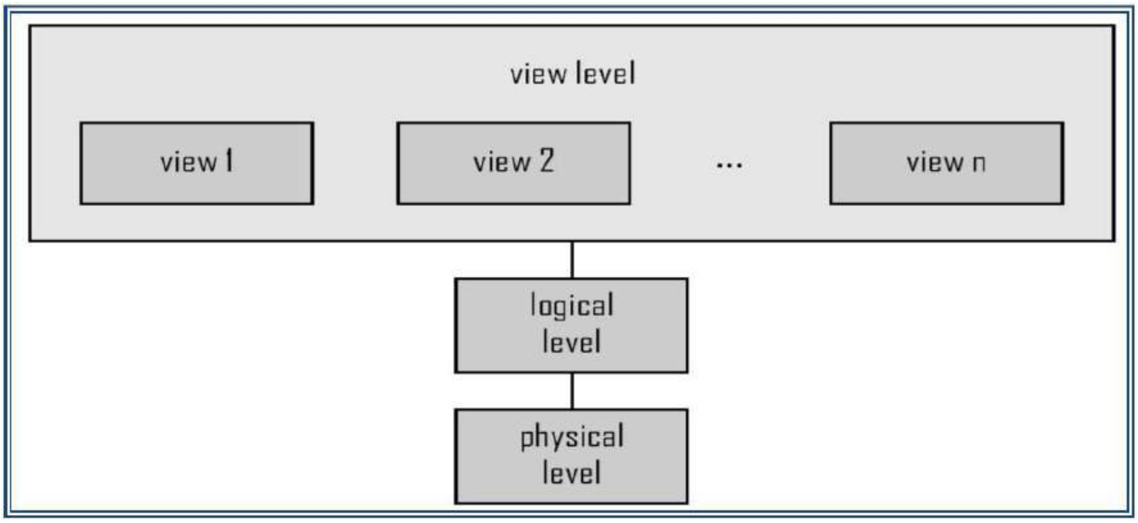
Crash recovery

Security and access control

View of Data

Architecture for a database system:

A database system is a collection of interrelated data and a set of programs that allow users to access and modify these data. The main task of database system is to provide abstract view of data i.e hides certain details of storage to the users.



Data Abstraction:

Major purpose of dbms is to provide users with abstract view of data i.e. the system hides cert ain details of how the data are stored and maintained. Since database system users are not computer trained,developers hide the complexity from users through 3 levels of abstraction , to simplify user's interaction with the system.

Levels of Abstraction

Physical level of data abstraction: Describes how a record (e.g., customer) is stored. This is the lowest level of abstraction which describes how data are actually stored.

Logical level of data abstraction: The next highest level of abstraction which hides what data are actually stored in the database and what relations hip exists among them. Describes data stored in database, and the relationships among the data.

```
type customer = record;
customer_id:string;
customer_name:string;
```

```
customer_stree:string;
customer_city :   string;
end;
```

View Level of data abstraction: The highest level of abstraction provides security mechanism to prevent user from accessing certain parts of database. Application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes and to simplify the interaction with the system.

Summary

DBMS used to maintain, query large datasets.

Benefits include recovery from system crashes, concurrent access, quick application development, data integrity and security.

Levels of abstraction give data independence.

A DBMS typically has a layered architecture.

DBAs hold responsible jobs and are well-paid!

DBMS R&D is one of the broadest, most exciting areas in CS.

Instances and Schemas:

Similar to types and variables in programming languages. Database changes over time when information is inserted or deleted.

Instance – the actual content of the database at a particular point in time analogous to the value of a variable is called an instance of the database.

Schema – the logical structure of the database called the database schema. Schema is of three types: Physical schema, logical schema and view schema.

Example: The database consists of information about a set of customers and accounts and the relationship between them)Analogous to type information of a variable in a program

Physical schema: Database design at the physical level is called physical schema. How the data stored in blocks of storage is described at this level.

Logical schema: database design at the logical level Instances and schemas, programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level.

View schema: Design of database at view level is called view schema**.** This generally describes end user interaction with database systems.

Physical Data Independence – The ability to modify the physical schema without changing the logical schema.

Applications depend on the logical schema

In general, the interfaces between the various levels and components should be well defined so

that changes in some parts do not seriously influence others.

Example: University Database

Conceptual schema:

Students(sid: string, name: string, login: string, age: integer, gpa:real)

Courses(cid: string, cname:string, credits:integer)

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

Physical schema: Relations stored as unordered files.

Index on first column of Students.

External Schema (View):

Course_info(cid:string,enrollment:integer)

Data Independence:

Applications insulated from how data is structured and stored.

<u>**Logical data independence**</u>: Protection from changes in *logical* structure of data.

<u>Physical data independence</u>: Protection from changes in *physical* structure of data.

History of Database Systems:

1950s and early 1960s:

–Data processing using magnetic tapes for storage

Tapes provide only sequential access

–Punched cards for input

Late 1960s and 1970s:

–Hard disks allow direct access to data

–Network and hierarchical data models in widespread use

–Ted Codd defines the relational data model

Would win the ACM Turing Award for this work

IBM Research begins System R prototype

UC Berkeley begins Ingres prototype

–High-performance (for the era) transaction processing

1980s:

–Research relational prototypes evolve into commercial systems

SQL becomes industry standard

–Parallel and distributed database systems

–Object-oriented database systems

1990s:

–Large decision support and data-mining applications

–Large multi-terabyte data warehouses

–Emergence of Web commerce

2000s:

–XML and XQuery standards

–Automated database administration

–Increasing use of highly parallel database systems

–Web-scale distributed data storage systems

Data Models:

A Data Model is a logical structure of Database. It is a collection of concepts for describing data, reflects entities, attributes, relationship among data, constrains etc. A schema is a description of a particular collection of data, using the given data model. The relational model of data is the most widely used model today. it is a collection of tools for describing

- Data
- Data relationships
- Data semantics
- Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi structured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

Every relation has a *schema*, which describes the columns, or fields.

Different types of data models are:

Relational model: The relational model uses a collection of tables to represent both data and relationships among those data. Each table has multiple columns with unique name.

- It is example of record based model

– These models are structured is fixed-format of several types.
– Each table contains records of particular type
– Each record type defines fixed number of fields, or attributes.
– The columns of the table correspond to attributes of the record type.

The relational data model is the most widely used data model and majority of current database systems are based on relational model.

Entity-relationship model: The E-R model is based on a perception of real world that consists of basic objects called entities and relationships among these objects. An entity is a 'thing' or 'object' in the real world, E-R model is widely used in database design.

Introduction to Database Design:

Conceptual design: *(ER Model is used at this stage.)*

–What are the *entities* and *relationships* in the enterprise?

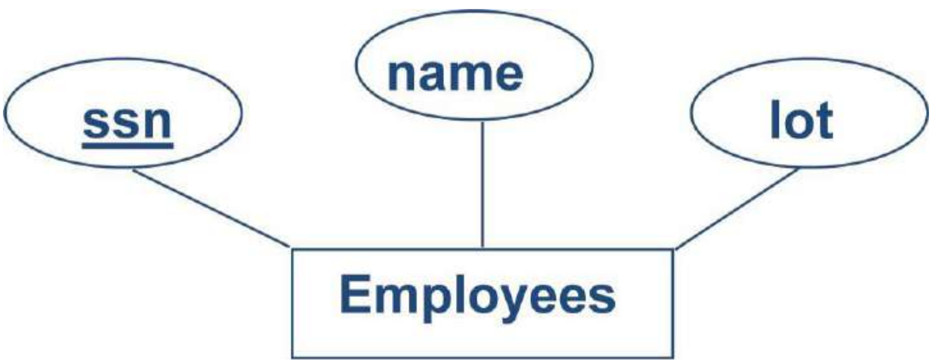
–What information about these entities and relationships should we store in the database?

–What are the *integrity constraints* or *business rules* that hold?

–A database `schema' in the ER Model can be represented pictorially (*ER diagrams*).

–Can map an ER diagram into a relational schema.

ER Model:



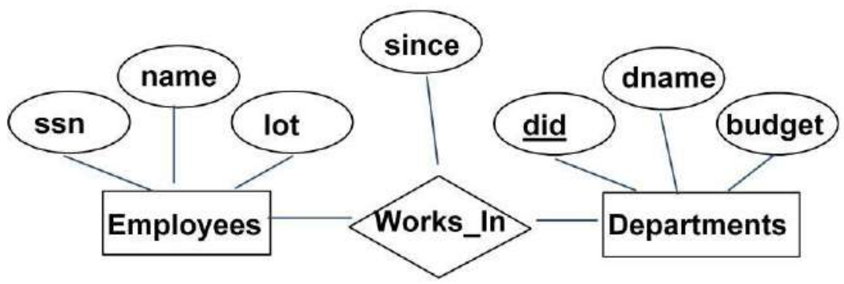
Entity: Real-world object distinguishable from other objects. An entity is described (in DB) using a set of *attributes*.

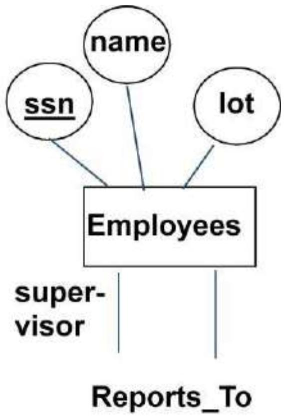
Entity Set: A collection of similar entities. E.g., all employees.

– All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)

– Each entity set has a *key*.

– Each attribute has a *domain*.





Relationship: Association among two or more entities. E.g., Attishoo works in Pharmacy department.

Relationship Set: Collection of similar relationships.

–An n-ary relationship set R relates n entity sets E1 ... En; each relationship in R involves entities e1 E1, ..., en En

Same entity set could participate in different relationship sets, or in different "roles" in same set.

Modeling:

A *database* can be modeled as:

–a collection of entities,

–relationship among entities.

Entities and Entity Sets:

An **entity** is an object that exists and is distinguishable from other objects.

Example: specific person, company, event, plant

Entities have *attributes*

Example: people have *names* and *addresses*

An **entity set** is a set of entities of the same type that share the same properties.

Example: set of all persons, companies, trees, holidays

Example:Entity Sets customer and loan

321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

customer

L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

loan

Attributes:

An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

Domain – the set of permitted values for each attribute

Attribute types:

–*Simple* and *composite* attributes.

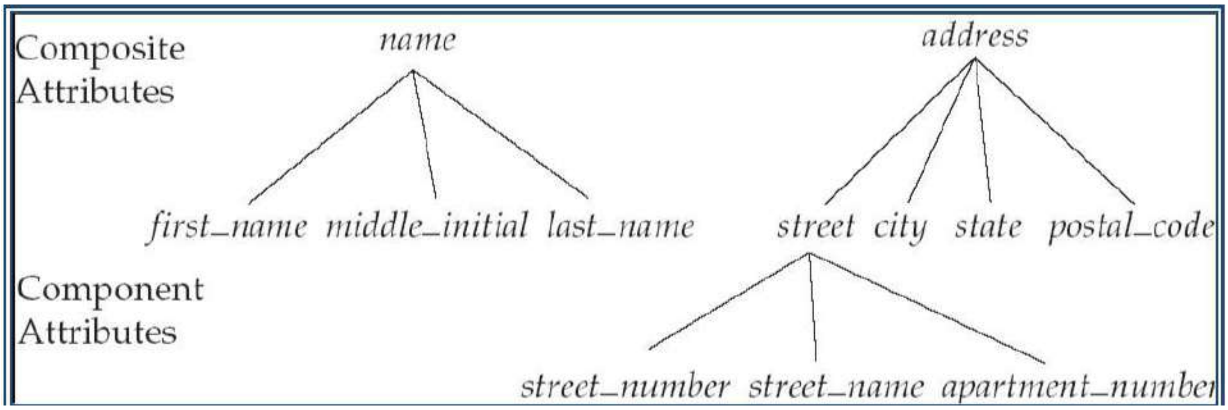
–*Single-valued* and *multi-valued* attributes Example:

multivalued attribute: *phone_numbers*

–*Derived* attributes can be computed from other attributes

Example: age, given date_of_birth

Composite Attributes



Mapping Cardinality Constraints

Express the number of entities to which another entity can be associated via a relationship set.

Most useful in describing binary relationship sets.

For a binary relationship set the mapping cardinality must be one of the following types:

–One to one

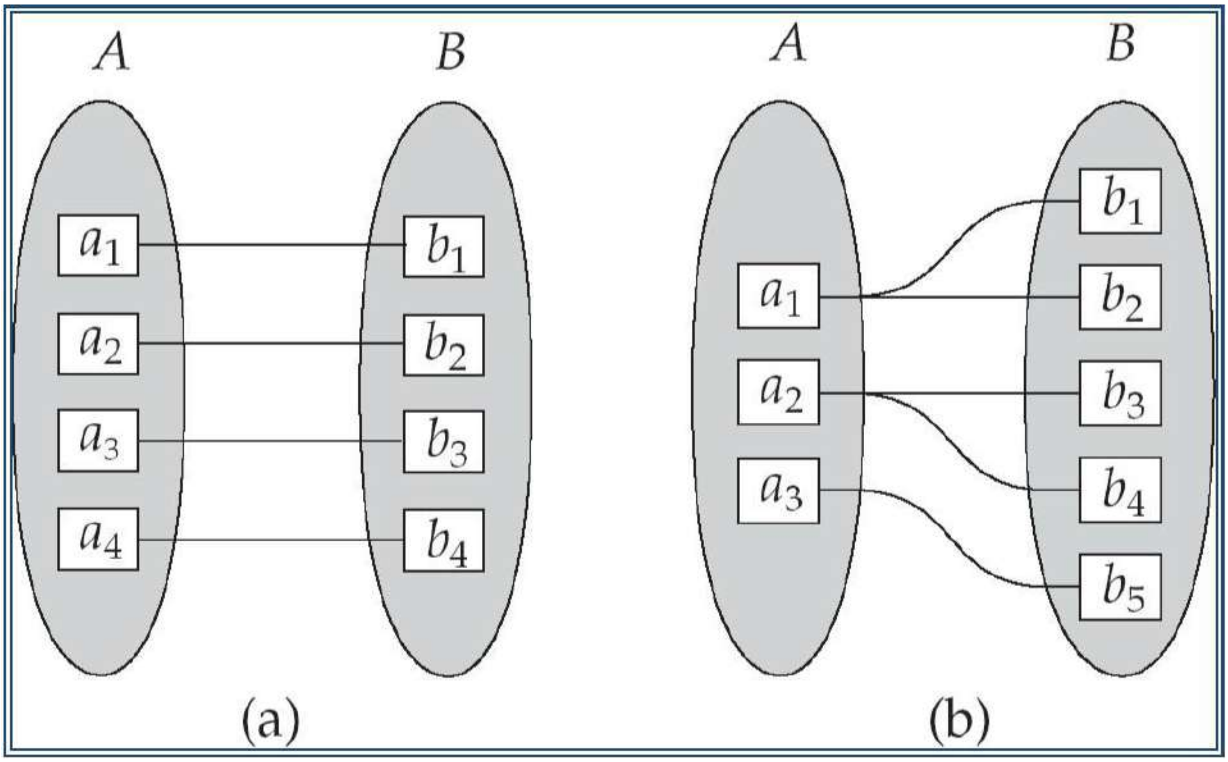
–One to many

–Many to one

–Many to many

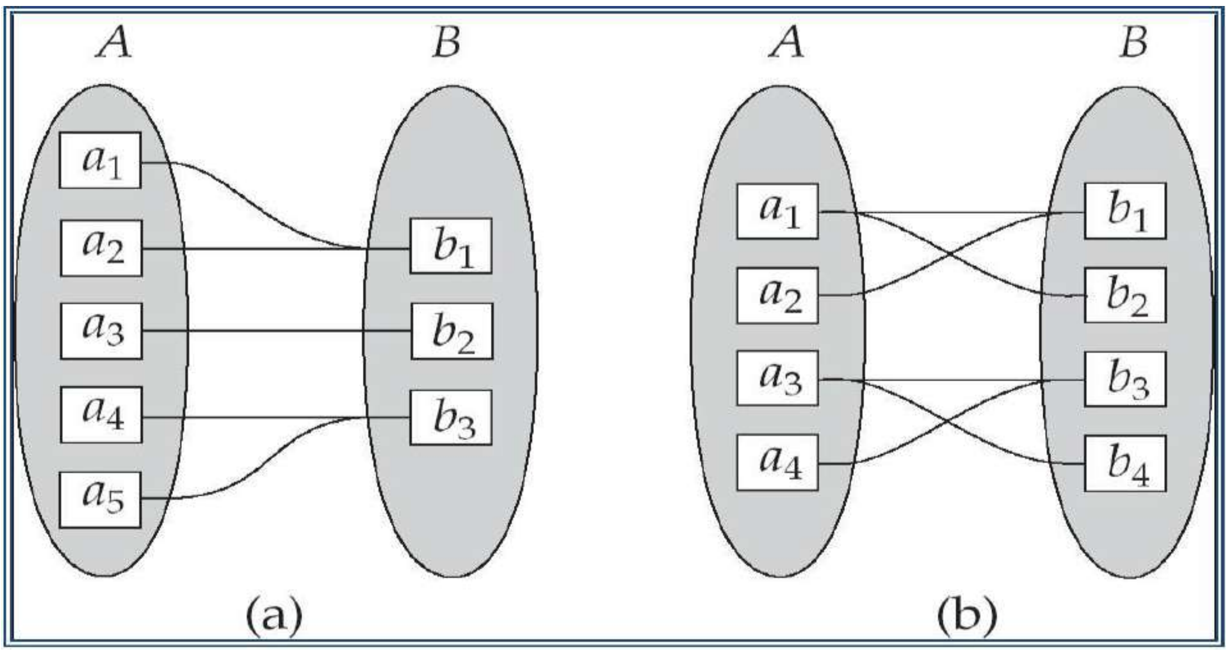
Mapping Cardinalities:

Note: Some elements in *A* and *B* may not be mapped to any elements in the other set



Mapping Cardinalities

Note: Some elements in A and B may not be mapped to any elements in the other set



Relationships and Relationship Sets

A **relationship** is an association among several entities

A **relationship set** is a mathematical relation among $n \geq 2$ entities, each taken from

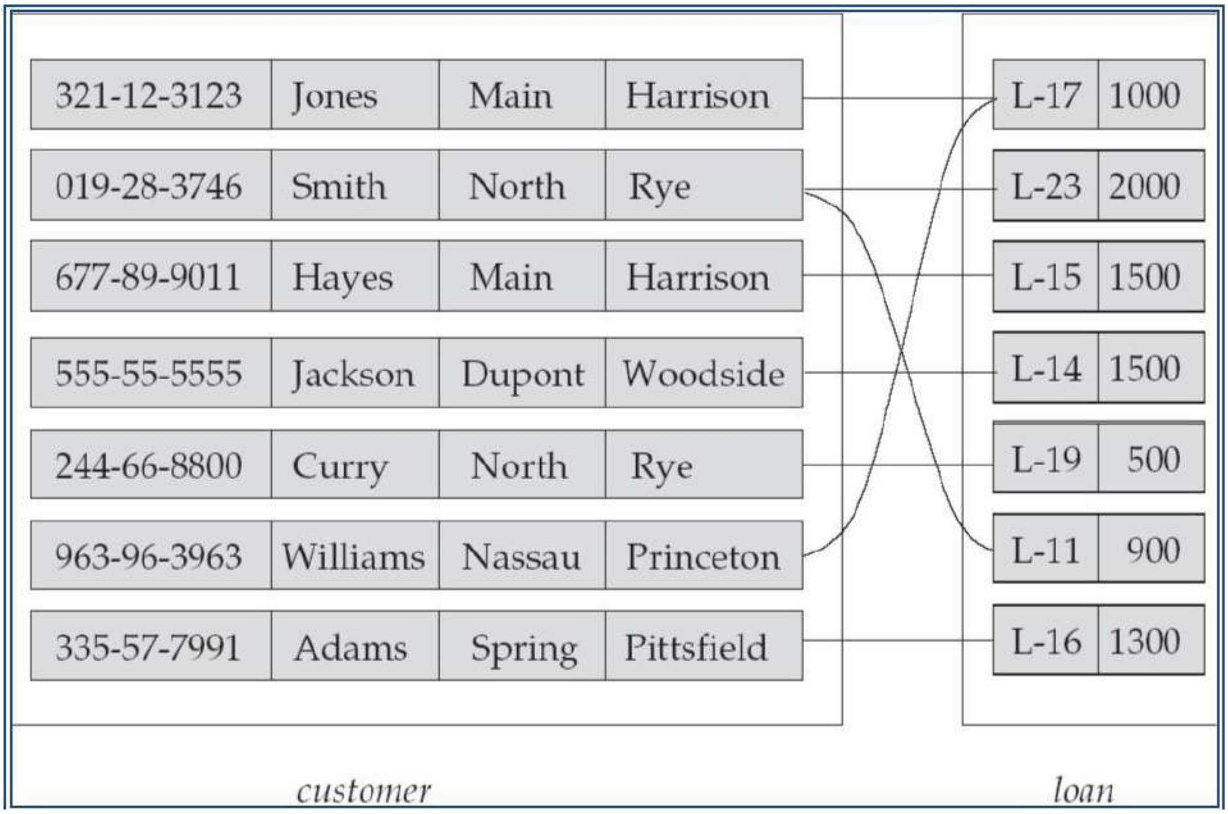
entity sets

$\{(e_1, e_2, \dots e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$ where $(e_1, e_2, \dots, e_n)$ is a relationship

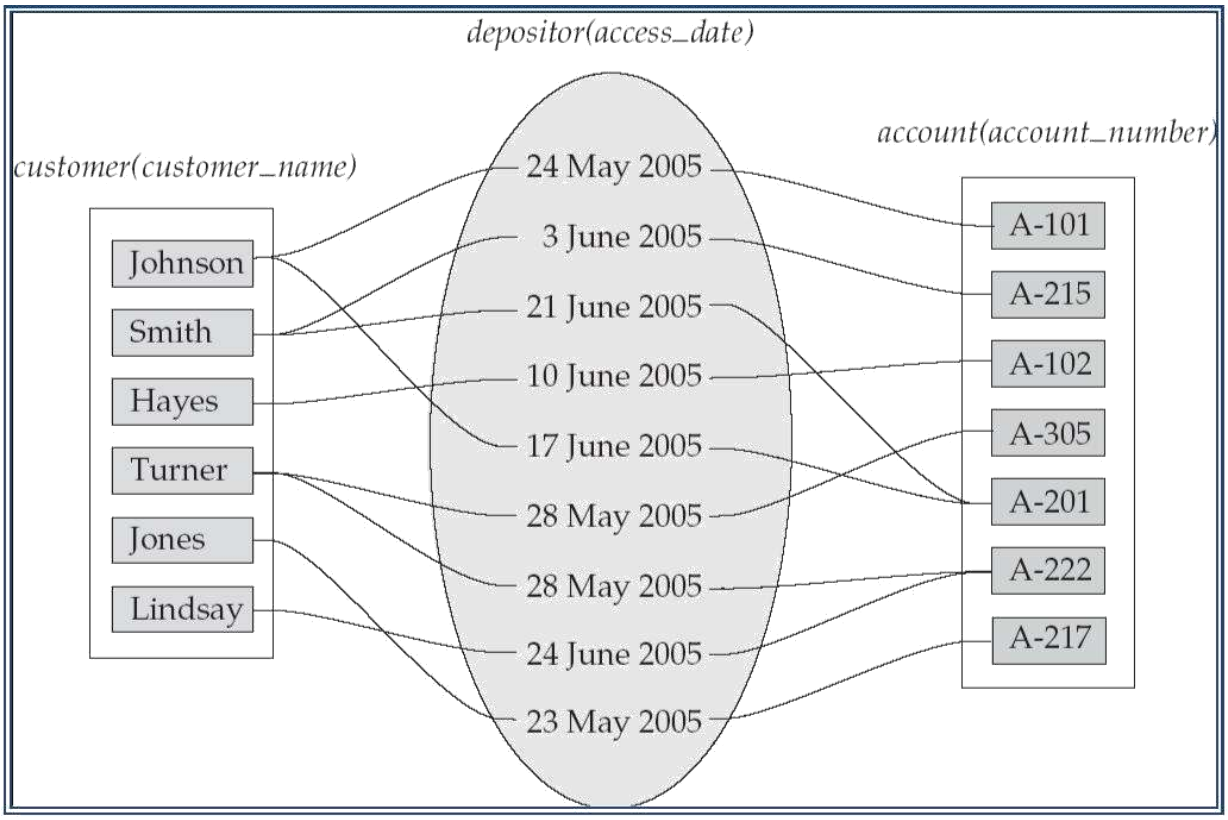
– Example:

(Hayes, A-102) $\in$ *depositor*

Relationship Set *borrower*



An **attribute** can also be property of a relationship set.



For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date*

Degree of a Relationship Set

Refers to number of entity sets that participate in a relationship set.

Relationship sets that involve two entity sets are **binary** (or degree two). Generally, most relationship sets in a database system are binary.

Relationship sets may involve more than two entity sets.

Example: Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee, job, and branch*

Relationships between more than two entity sets are rare. Most relationships are binary.

Weak Entities

A *weak entity* can be identified uniquely only by considering the primary key of another (*owner*) entity.

- Owner entity set and weak entity set must participate in a one-to-many relationship set (one owner, many weak entities).
- Weak entity set must have total participation in this *identifying* relationship set.

Weak Entity Sets

An entity set that does not have a primary key is referred to as a **weak entity set**.

The existence of a weak entity set depends on the existence of a **identifying entity set**

- it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
- Identifying relationship depicted using a double diamond

The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

depict a weak entity set by double rectangles.

underline the discriminator of a weak entity set with a dashed line.

payment_number – discriminator of the *payment* entity set

Primary key for *payment* – (*loan_number, payment_number*)

Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.

If *loan_number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan_number* common to *payment* and *loan*

More Weak Entity Set Examples

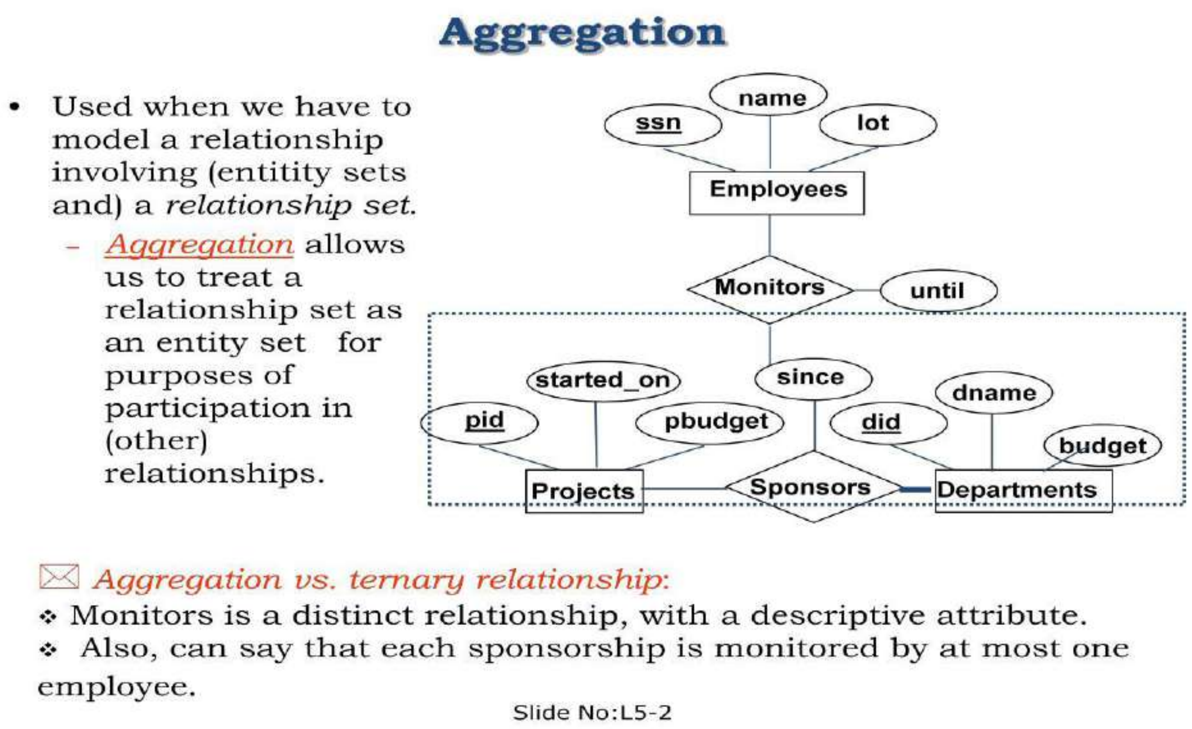
In a university, a *course* is a strong entity and a *course_offering* can be modeled as a weak entity

The discriminator of *course_offering* would be *semester* (including year) and *section_number* (if there is more than one section)

If we model *course_offering* as a strong entity we would model *course_number* as an attribute.

Then the relationship with *course* would be implicit in the *course_number* attribute

Aggregation



Relationship sets *works_on* and *manages* represent overlapping information

– Every *manages* relationship corresponds to a *works_on* relationship

However, some *works_on* relationships may not correspond to any *manages* relationships. So we can't discard the *works_on* relationship

Eliminate this redundancy via *aggregation*

– Treat relationship as an abstract entity

– Allows relationships between relationships

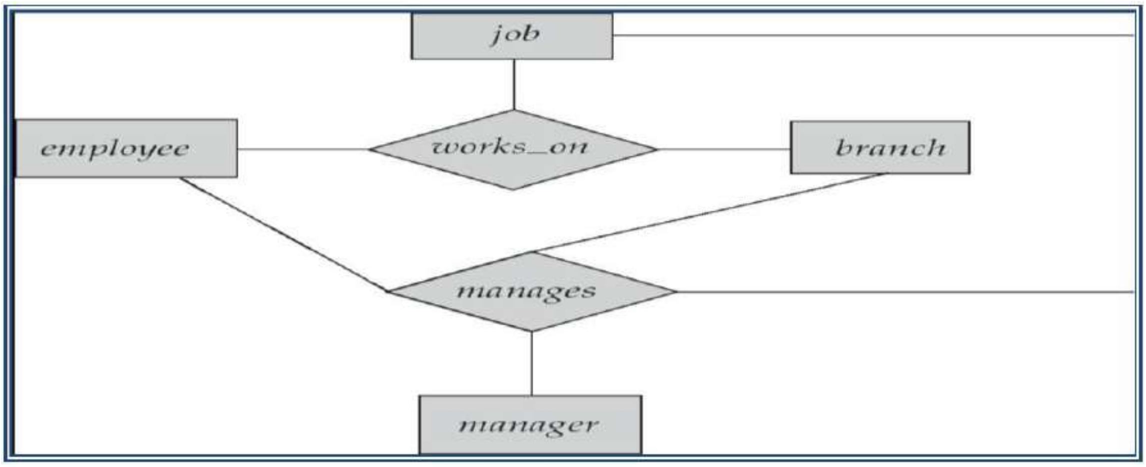
– Abstraction of relationship into new entity

Without introducing redundancy, the following diagram represents:

– An employee works on a particular job at a particular branch

– An employee, branch, job combination may have an associated manager

E-R Diagram with Aggregation



Conceptual Design with ER Model

<u>Design choices:</u>

– Should a concept be modeled as an entity or an attribute?

– Should a concept be modeled as an entity or a relationship?

– Identifying relationships: Binary or ternary? Aggregation?

Constraints in the ER Model:

– A lot of data semantics can (and should) be captured.

– But some constraints cannot be captured in ER diagrams.

Entity vs. Attribute

Should *address* be an attribute of Employees or an entity (connected to Employees by a relationship)?

Depends upon the use we want to make of address information, and the semantics of the data:

If we have several addresses per employee, *address* must be an entity (since attributes cannot be set-valued).

If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, *address* must be modeled as an entity (since attribute values are atomic).

An example in the other direction: a ternary relation Contracts relates entity sets Parts, Departments and Suppliers, and has descriptive attribute *qty*. No combination of binary . relationships is an adequate substitute:

S “can-supply” P, D “needs” P,and D “deals-with” S does not imply that D has agreed to buy P from S.

–How do we record *qty*?

Introduction to relational model

Relational Database: Definitions

Relational database: a set of *relations*

Relation: made up of 2 parts:

– *Instance* : a *table,* with rows and columns.

#Rows = *cardinality*, #fields = *degree* / *arity*.

– *Schema* : specifies name of relation, plus name and type of each column. E.G.

Students (*sid*: string, *name*: string, *login*: string, *age*: integer, *gpa*: real).

Can think of a relation as a *set* of rows or *tuples* (i.e., all rows are distinct).

Example Instance of Students Relation

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@eecs	18	3.2
53650	Smith	smith@math	19	3.8

Cardinality = 3, degree = 5, all rows distinct

Do all columns in a relation instance have to be distinct?

Relational Query Languages A major strength of the relational model: supports simple, powerful *querying* of data.

Queries can be written intuitively, and the DBMS is responsible for efficient evaluation.

– The key: precise semantics for relational queries.

– Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.

Creating Relations in SQL

Creates the Students relation. Observe that the type of each field is specified, and enforced by the DBMS whenever tuples are added or modified.

CREATE TABLE Students (sid CHAR(20), name CHAR(20),login CHAR(10),age: INTEGER, gpa: REAL)

As another example, the Enrolled table holds information about courses that students take.

CREATE TABLE Enrolled (sid: CHAR(20),cid: CHAR(20), grade: CHAR(2))

Integrity Constraints (ICs) over Relations:

IC: condition that must be true for *any* instance of the database; e.g., *domain constraints.*

ICs are specified when schema is defined.

ICs are checked when relations are modified.

A *legal* instance of a relation is one that satisfies all specified ICs.

DBMS should not allow illegal instances.

If the DBMS checks ICs, stored data is more faithful to real-world meaning.

– Avoids data entry errors, too!

Primary Key Constraints

A set of fields is a <u>*key*</u> for a relation if :

No two distinct tuples can have same values in all key fields, and

This is not true for any subset of the key.

– Part 2 false? A *superkey*.

– If there's >1 key for a relation, one of the keys is chosen (by DBA) to be the *primary key*.

E.g., *sid* is a key for Students. (What about *name*?) The set {*sid*, *gpa*} is a superkey.

Primary and Candidate Keys in SQL

Possibly many <u>*candidate keys*</u> (specified using UNIQUE), one of which is chosen as the *primary key*.

Foreign Keys, Referential Integrity

<u>*Foreign key*</u> : Set of fields in one relation that is used to `refer' to a tuple in another relation. (Must correspond to primary key of the second relation.) Like a `logical pointer'.

E.g. *sid* is a foreign key referring to Students:

Foreign Keys in SQL

Only students listed in the Students relation should be allowed to enroll for courses.

Enforcing Integrity constraints

Consider Students and Enrolled; *sid* in Enrolled is a foreign key that references Students.

What should be done if an Enrolled tuple with a non-existent student id is inserted? (*Reject it!*)

What should be done if a Students tuple is deleted?

– Also delete all Enrolled tuples that refer to it.

– Disallow deletion of a Students tuple that is referred to.

– Set sid in Enrolled tuples that refer to it to a *default sid*.

– (In SQL, also: Set sid in Enrolled tuples that refer to it to a special value *null*, denoting *`unknown'* or *`inapplicable'*.)

Similar if primary key of Students tuple is updated.

Referential Integrity in SQL

SQL/92 and SQL:1999 support all 4 options on deletes and updates.

– Default is NO ACTION (*delete/update is rejected*)

– CASCADE (also delete all tuples that refer to deleted tuple)

– SET NULL / SET DEFAULT (sets foreign key value of referencing tuple)

Where do ICs Come From?

ICs are based upon the semantics of the real-world enterprise that is being described in the database relations.

We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.

– An IC is a statement about *all possible* instances!

– From example, we know *name* is not a key, but the assertion that *sid* is a key is given to us.

Key and foreign key ICs are the most common; more general ICs supported too.

Data base Languages:

Data Control Language (DCL): It is used to control privilege in database. To perform any operations like creating tables, view and modifying we need privileges which are of two types.

System:- Creating session and tables are types of system privilege.

Object:- Any command or query to work on tables comes under object privilege.

DCL defines two commands GRANT and REVOKE.

GRANT:-Gives user access privilege to database.

REVOKE: - To take back permissions from users.

CONNECTING TO ORACLE:

CONNECT<USER NAME>/<PASSWORD>@<DATABASE NAME>;

Create user login:

CREATE USER <USER_NAME> IDENTIFIED BY <PASSWORD>;

Provide roles:

GRANT CONNECT, CREATE SESSION, RESOURCE TO <USER_NAME>;

Provide privileges:

GRANT ALL PRIVILEGES TO <USER_NAME>;

Data Definition Language (DDL):

Specification notation for defining the database schema by a set of definitions.
DDL compiler generates a set of tables stored in a *data dictionary*
Data dictionary contains metadata (i.e., data about data)
Database schema
Data *storage and definition* language
Specifies the storage structure and access methods used
Integrity constraints
Domain constraints
Referential integrity (e.g. *branch_name* must correspond to a valid to branch in the *branch* table)
Authorization
Procedural – user specifies what data is required and how to get those data
Declarative (nonprocedural) – user specifies what data is required without specifying how to get those data.

DDL: Data Definition Language

All DDL commands are auto-committed. That means it saves all the changes permanently in the database.

Command	Description
create	to create new table or database
alter	for alteration
truncate	delete data from table
drop	to drop a table
rename	to rename a table

CREATE command:

create is a DDL command used to create a table or a database.

Creating a database

To create a database in RDBMS, *create* command is uses. Following is the Syntax,

Create database database-name;

Example for creating database

Create database *Test*;

The above command will create a database named **Test**.

Creating a table

create command is also used to create a table. We can specify names and datatypes of various columns along. Following is the Syntax,

create table *table-name*

{

Column-name1 *datatype1*,

Column-name2 *datatype2*,

Column-name1 *datatype3*,

Column-name2 *datatype4*

};

Create table command will tell the database system to create a new table with given table name and column information.

Example for creating table

Create table *Student*(id *int*, name *varchar*, age *int*);

The above command will create a new table **Student** in database system with 3 columns, namely id, name and age.

ALTER command

alter command is used for alteration of table structures. There are various uses of *alter* command, such as,

- to add a column to existing table
- to rename any existing column
- to change datatype of any column or to modify its size.
- *alter* is also used to drop a column.

To add column to existing table

Using alter command we can add a column to an existing table. Following is the Syntax,

Alter table *table-name* add(column-name *datatype*);

Here is an Example for this,

Alter table *student* add(address *char*);

The above command will add a new column *address* to the **Student** table

To add multiple column to existing table

Using alter command we can even add multiple columns to an existing table. Following is the Syntax,

Alter table *table-name* add(column1 *datatype1*, column2 *datatype2*, column3 *datatype3*, colum4 *datatype4*);

Here is an Example for this,

Alter table *student* add(father_name *varchar(60)*, mother_name v*archar(60)*, DOB *date*);

Date input format is:- ’date-month-year’ i.e ’10-jan-2016’

The above command will add three new columns to the **Student** table

---

To add column with default value

alter command can add a new column to an existing table with default values. Following is the Syntax,

alter table table_name add (column_name *datatype* default *data*);

Here is an Example for this,

alter table Student add(branch *char* default 'CSE');

The above command will add a new column with default value to the **Student** table

---

To modify an existing column

alter command is used to modify data type of an existing column . Following is the Syntax,

alter table *table-name* modify(column-name *datatype*);

Here is an Example for this,

alter table *Student* modify(address *varchar(30)*);

The above command will modify *address* column of the **Student table**

To rename a column

Using alter command you can rename an existing column. Following is the Syntax,

alter table *table-name* rename *old-column-name* to new-*column-name*;

Here is an Example for this,

alter table *Student* rename address to Location;

The above command will rename *address* column to *Location*.

To drop a column

alter command is also used to drop columns also. Following is the Syntax,

alter table *table-name* drop(column-name);

Here is an Example for this,

alter table *Student* drop(address);

The above command will drop *address* column from the **Student table**

SQL queries to Truncate, Drop or Rename a Table

truncate command

truncate command removes all records from a table. But this command will not destroy the table's structure. When we apply truncate command on a table its Primary key is initialized. Following is its Syntax,

truncate table *table-name*

Here is an Example explaining it.

truncate table *Student*;

The above query will delete all the records of **Student** table.

truncate command is different from **delete** command. delete command will delete all the rows from a table whereas truncate command re-initializes a table(like a newly created table).

eg. If you have a table with 10 rows and an auto_increment primary key, if you use *delete* command to delete all the rows, it will delete all the rows, but will not initialize the primary key, hence if you will insert any row after using delete command, the auto_increment primary key will start from 11. But in case of *truncate*command, primary key is re-initialized.

---

drop command

drop query completely removes a table from database. This command will also destroy the table structure. Following is its Syntax,

drop table *table-name;*

Here is an Example explaining it.

drop table *Student*;

The above query will delete the **Student** table completely. It can also be used on Databases. For Example, to drop a database,

drop database *Test*;

The above query will drop a database named **Test** from the system.

---

rename query

rename command is used to rename a table. Following is its Syntax,

rename table *old-table-name* to *new-table-name*

Here is an Example explaining it.

rename table *Student* to *Student-record*;

The above query will rename **Student** table to **Student-record**.

DML COMMANDS:

INSERT command

Insert command is used to insert data into a table. Following is its general syntax,

insert into *table_name* values(data1,data2,…….);

Lets see an example,

Consider a table **Student** with following fields.

S_id	S_Name	age

INSERT into *Student* values(101,'Adam',15);

The above command will insert a record into **Student** table.

S_id	S_Name	age
101	Adam	15

Example to Insert NULL value to a column

Both the statements below will insert NULL value into **age** column of the Student table.

INSERT into *Student*(id,name) values(102,'Alex');

Or,

INSERT into *Student* values(102,'Alex',null);

The above command will insert only two column value other column is set to null.

S_id	S_Name	age
101	Adam	15
102	Alex	

Example to Insert Default value to a column

INSERT into *Student* values(103,'Chris',default);

S_id	S_Name	age
101	Adam	15
102	Alex	
103	chris	14

Suppose the **age** column of student table has default value of 14.

Also, if you run the below query, it will insert default value into the age column, whatever the default value may be.

INSERT into *Student* values(103,'Chris');

UPDATE command

Update command is used to update a row of a table. Following is its general syntax,

UPDATE *table-name* set column-name = value *where* condition;

Lets see an example,

update *Student* set age=18 where s_id=102;

S_id	S_Name	age
101	Adam	15
102	Alex	18
103	chris	14

Example to Update multiple columns

UPDATE *Student* set s_name='Abhi',age=17 where s_id=103;

The above command will update two columns of a record.

S_id	S_Name	age
101	Adam	15
102	Alex	18
103	Abhi	17

---

3) Delete command

Delete command is used to delete data from a table. Delete command can also be used with condition to delete a particular row. Following is its general syntax, **DELETE from *table-name*;**

Example to Delete all Records from a Table

DELETE from *Student*;

The above command will delete all the records from **Student** table.

Example to Delete a particular Record from a Table

Consider the following **Student** table

S_id	**S_Name**	**age**
101	Adam	15
102	Alex	18
103	Abhi	17

DELETE from *Student* where s_id=103;

The above command will delete the record where s_id is 103 from **Student** table.

S_id	**S_Name**	**age**
101	Adam	15
102	Alex	18

TCL command

Transaction Control Language(TCL) commands are used to manage transactions in database.These are used to manage the changes made by DML statements. It also allows statements to be grouped together into logical transactions.

Commit command

Commit command is used to permanently save any transaaction into database.

Following is Commit command's syntax,

commit;

Rollback command

This command restores the database to last commited state. It is also use with savepoint command to jump to a savepoint in a transaction. Following is Rollback command's syntax,

rollback to *savepoint-name*;

Savepoint command

savepoint command is used to temporarily save a transaction so that you can rollback to that point whenever necessary.

Following is savepoint command's syntax,

savepoint *savepoint-name*;

Example of Savepoint and Rollback

Following is the **class** table,

ID	NAME
1	abhi
2	adam
4	alex

Lets use some SQL queries on the above table and see the results.

INSERT into *class* values(5,'Rahul');

commit;

UPDATE *class* set name='abhijit' where id='5'; savepoint A;

INSERT into *class* values(6,'Chris');

savepoint B;

INSERT into *class* values(7,'Bravo');

savepoint C;

SELECT * from *class*;

The resultant table will look like,

Now **rollback** to **savepoint B**

rollback to B;

SELECT * from *class*;

The resultant table will look like

Now **rollback** to **savepoint A**

rollback to A;

SELECT * from *class*;

The result table will look like

DCL command

Data Control Language(DCL) is used to control privilege in Database. To perform any operation in the database, such as for creating tables, sequences or views we need privileges. Privileges are of two types,

System : creating session, table etc are all types of system privilege.

Object : any command or query to work on tables comes under object privilege. DCL defines two commands,

Grant : Gives user access privileges to database.

Revoke : Take back permissions from user.

To Allow a User to create Session

grant create session to *username*;

To Allow a User to create Table

grant create table to *username*;

To provide User with some Space on Tablespace to store Table

alter user *username* quota unlimited on system;

To Grant all privilege to a User

grant sysdba to *username*

To Grant permission to Create any Table

grant *create* any table to *username*

To Grant permission to Drop any Table

grant *drop* any table to *username*

To take back Permissions

revoke create table from *username*

Data Base Access from Application Programs:

SQL: Application programs generally access databases through one of

–Language extensions to allow embedded SQL

–Application program interface (e.g., ODBC/JDBC) which allow SQL queries to

be sent to a database.

Customer:

Example: Find the name of the customer with customer-id 192-83-7465
SQL>select *customer.customer_name*

Example: Find the balances of all accounts held by the customer with customer-Id 192-83-7465.

SQL>select *account.balance*
from *depositor,account*
where *depositor.customer_id*=‘192-83-7465’and
depositor.account_number = account.account_number;

Database Architecture:

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

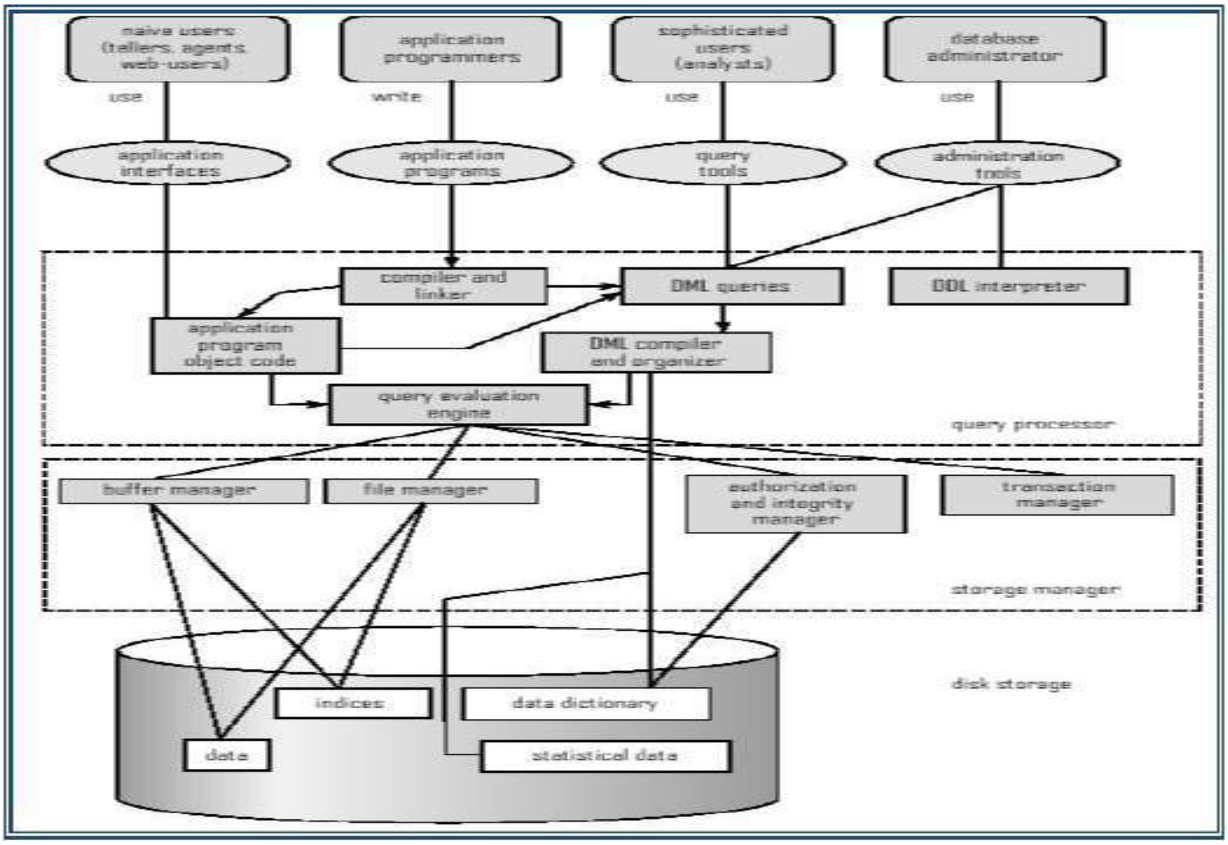
Centralized

Client-server

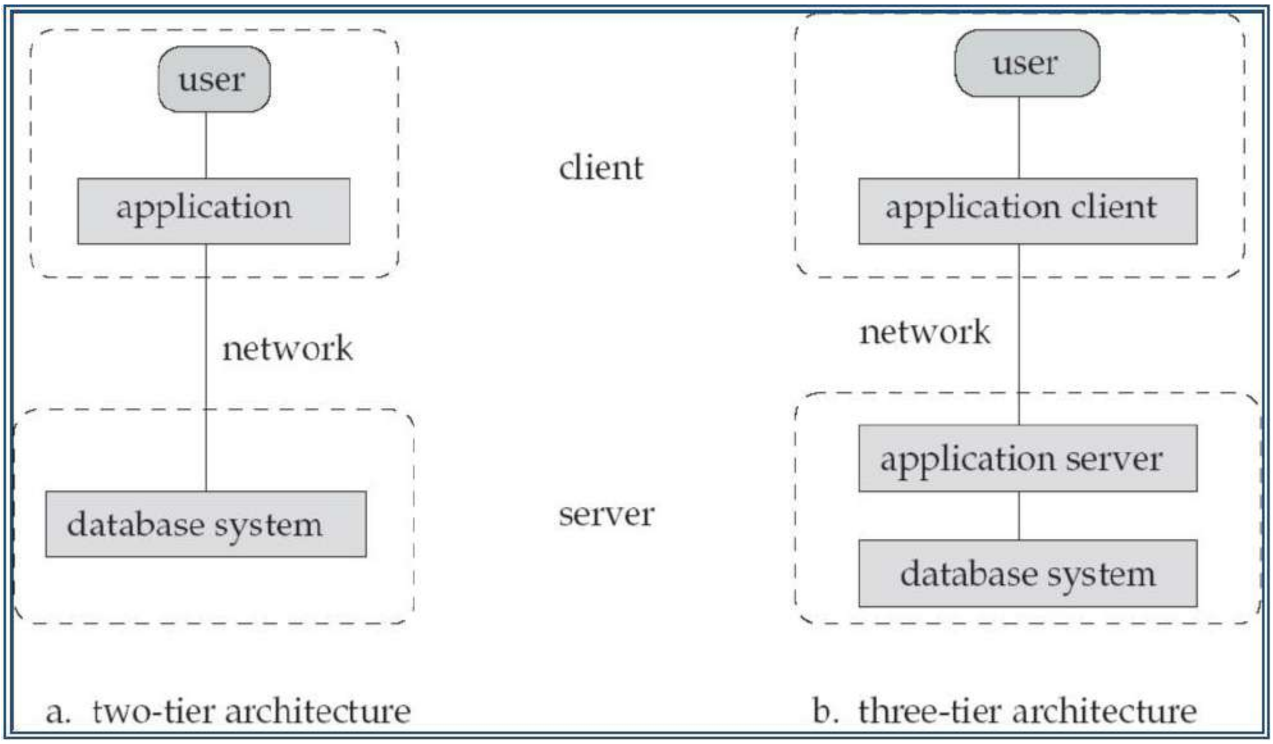
Parallel (multiple processors and disks)

Distributed

Overall System Structure



Database Application Architectures:



Transaction Management:

A **transaction** is a collection of operations that performs a single logical function in a database application A transaction in a database system must maintain atomicity, consistency, isolation, and durability − commonly known as ACID properties properties − in order to ensure accuracy, completeness, and data integrity.

Transaction-management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

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 the database system are

- Storage management
- Query processing
- Transaction processing

Storage Management

Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The storage manager is responsible to the following tasks:

- Interaction with the file manager
- Efficient storing, retrieving and updating of data
- Authorization and integrity manager
- Integrity
- Transaction manager
- File manager
- Buffer manager

Issues:

– Storage access

– File organization

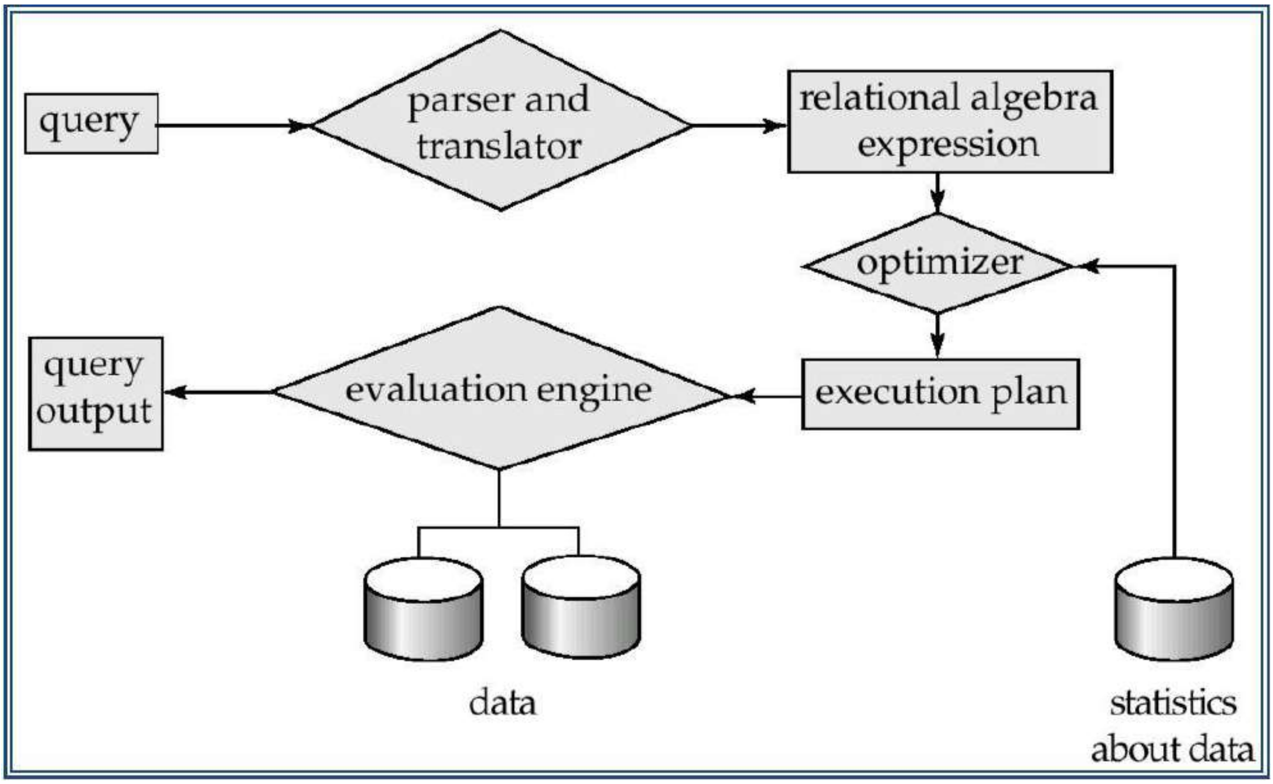
– Indexing and hashing

Query Processing

Parsing and translation

Optimization

Evaluation



Alternative ways of evaluating a given query

– Equivalent expressions

– Different algorithms for each operation

Cost difference between a good and a bad way of evaluating a query can be enormous

Need to estimate the cost of operations

– Depends critically on statistical information about relations which the database must maintain

– Need to estimate statistics for intermediate results to compute cost of complex expressions

Database Users and Administrators:

Database Users

Users are differentiated by the way they expect to interact with the system

Application programmers – interact with system through DML calls

Sophisticated users – form requests in a database query language

Specialized users – write specialized database applications that do not fit into the traditional data processing framework

Naïve users – invoke one of the permanent application programs that have been written previously

– Examples, people accessing database over the web, bank tellers, clerical staff

Database Administrator

Coordinates all the activities of the database system

–has a good understanding of the enterprise's information resources and needs.

Database administrator's duties include:

–Storage structure and access method definition

–Schema and physical organization modification

–Granting users authority to access the database

–Backing up data

–Monitoring performance and responding to changes

–Database tuning.

UNIT-II

Relational Approach

Relational Algebra

Operations

Query examples

Relational Calculus

Tuple Relational Calculus

Domain Relational Calculus

Expressive power of algebra and calculus

Relational Algebra

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - *Cross-product* ($\times$) Allows us to combine two relations.
  - *Set-difference* ( ) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples ~~in~~ reln. 1 and in reln. 2.
- Additional operations:
  - Intersection, *join*, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Slide No:L6-4

Basic operations:

– *Selection* ( ) Selects a subset of rows from relation.

– *Projection* ( ) Deletes unwanted columns from relation.

– *Cross-product* ( ) Allows us to combine two relations.

– *Set-difference* ( ) Tuples in reln. 1, but not in reln. 2.

– *Union* ( ) Tuples in reln. 1 and in reln. 2.

Additional operations:

– Intersection, *join*, division, renaming: Not essential, but (very!) useful.

Since each operation returns a relation, operations can be *composed*! (Algebra is "closed".)

Projection

- Deletes attributes that are not in *projection list*.
- *Schema* of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.
- Projection operator has to eliminate *duplicates*! (Why??)
  - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$$\pi_{sname,rating}(S2)$$

age
35.0
55.5

$$\pi_{age}(S2)$$

Slide No:L6-5

Deletes attributes that are not in *projection list*.

Schema of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation.

Projection operator has to eliminate *duplicates*! (Why??)

– Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

Selection

- Selects rows that satisfy *selection condition*.
- No duplicates in result! (Why?)
- *Schema* of result identical to schema of (only) input relation.
- *Result* relation can be the *input* for another relational algebra operation! (*Operator composition.*)

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Slide No:L6-6

Selects rows that satisfy *selection condition*.

No duplicates in result! (Why?)

Schema of result identical to schema of (only) input relation.

Result relation can be the *input* for another relational algebra operation! (*Operator composition.*)

Set Operations:

Union, Intersection, Set-Difference

All of these operations take two input relations, which must be <u>*union-compatible*</u>:

– Same number of fields.

– `Corresponding' fields have the same type.

What is the *schema* of result?

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be *union-compatible*:
  - Same number of fields.
  - `Corresponding' fields have the same type.
- What is the *schema* of result?

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

Slide No:L6-7

Cross-Product

Each row of S1 is paired with each row of R1.

Result schema has one field per field of S1 and R1, with field names `inherited' if possible.

– *Conflict*: Both S1 and R1 have a field called *sid*.

Cross-Product

- Each row of S1 is paired with each row of R1.
- *Result schema* has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

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

- Renaming operator: $\rho\,(C(1 \to sid1, 5 \to sid2),\ S1 \times R1)$

Slide No:L6-8

Joins

$$R \bowtie_c S = \sigma_c (R \times S)$$

- Condition Join:

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- *Result schema* same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a *theta-join*.

Condition Join:

Result schema same as that of cross-product.

Fewer tuples than cross-product, might be able to compute more efficiently

Sometimes called a *theta-join*.

Equi-Join: A special case of condition join where the condition *c* contains only ***equalities***.

Result schema similar to cross-product, but only one copy of fields for which equality is specified.

Natural Join: Equijoin on *all* common fields.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

  *Find sailors who have reserved **all** boats.*
- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\left\{ \langle x \rangle \mid \exists \langle x, y \rangle \in A \;\; \forall \langle y \rangle \in B \right\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every* *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and $x \cup y$ is the list of fields of *A*.

Slide No:L6-11

Examples of Division A/B

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

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Slide No:L6-12

Find names of sailors who've reserved boat #103

Solution 1:

Find names of sailors who've reserved a red boat

Information about boat color only available in Boats; so need an extra join:

Find sailors who've reserved a red or a green boat

Can identify all red or green boats, then find sailors who've reserved one of these boats:

Find sailors who've reserved a red <u>and</u> a green boat

Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

Relational Calculus:

Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).

Calculus has *variables, constants, comparison ops, logical connectives* and *quantifiers*.

– *TRC*: Variables range over (i.e., get bound to) *tuples*.

– *DRC*: Variables range over *domain elements* (= field values).

– Both TRC and DRC are simple subsets of first-order logic.

Expressions in the calculus are called *formulas*. An answer tuple is essentially an assignment of constants to variables that make the formula evaluate to *true*.

Tuple Relational Calculus:

TRC – a declarative query language

TRC Formulas

Atomic expressions are the following:

r (t) -- true if t is a tuple in the relation instance r

t1. Ai t2 .Aj compOp is one of {, ≥, =, ≠ }

t.Ai c c is a constant of appropriate type

Composite expressions:

Any atomic expression
F1 ∧ F2 ,, F1 ∨ F2 , ¬ F1 where F1 and F2 are expressions

(∀t) (F), (∃t) (F) where F is an expression and t is a tuple variable Free Variables

Bound Variables – quantified variables

Obtain the rollNo, name of all girl students in the Maths Dept

{s.rollNo,s.name | student(s) ^ s.sex=‘F’ ^ (∃ d)(department(d) ^ d.name=‘Maths’ ^ d.deptId = s.deptNo)}

s: free tuple variable

d: existentially bound tuple variable

Determine the departments that do not have any girl students

student (rollNo, name, degree, year, sex, deptNo, advisor) department (deptId, name, hod, phone)

{d.name|department(d) ^ ¬(∃ s)(student(s) ^ s.sex =‘F’ ^ s.deptNo = d.deptId)

Obtain the names of courses enrolled by student named Mahesh

{c.name | course(c) ^ (∃s) (∃e) (student(s) ^ enrollment(e) ^ s.name = “Mahesh” ^ s.rollNo = e.rollNo ^ c.courseId = e.courseId }

Get the names of students who have scored ‘S’ in all subjects they have enrolled. Assume that every student is enrolled in at least one course.

{s.name | student(s) ^ (∀e)((enrollment(e) ^ e.rollNo = s.rollNo) → e.grade =‘S’)}

Get the names of students who have taken at least one course taught by their advisor

{s.name | student(s) ^ (∃e)(∃t)(enrollment(e) ^ teaching(t) ^ e.courseId = t.courseId ^ e.rollNo = s.rollNo ^ t.empId = s.advisor}

Domain Relational Calculus:

Query has the form:

DRC Formulas

Atomic formula:

– , or X *op* Y, or X *op* constant

– *op* is one of

Formula:

– an atomic formula, or

– , where p and q are formulas, or

– , where variable X is *free* in p(X), or

– , where variable X is *free* in p(X)

- The use of quantifiers and is said to *bind* X.

– A variable that is not bound is free.

Free and Bound Variables

- The use of quantifiers and in a formula is said to *bind* X.

– A variable that is not bound is free.

Let us revisit the definition of a query:

Find all sailors with a rating above 7

The condition ensures that the domain variables *I, N, T* and *A* are bound to fields of the same Sailors tuple.

- The term to the left of `|' (which should be read as *such that*) says that every tuple that satisfies *T*>7 is in the answer.

Modify this query to answer:

– Find sailors who are older than 18 or have a rating under 9, and are called 'Joe'.

Find sailors rated > 7 who have reserved boat #103

We have used as a shorthand for

Note the use of to find a tuple in Reserves that `joins with' the Sailors tuple under consideration.

Find sailors rated > 7 who've reserved a red boat

Observe how the parentheses control the scope of each quantifier's binding.

This may look cumbersome, but with a good user interface, it is very intuitive. (MS Access, QBE)

Find sailors who've reserved all boats

- Find all sailors *I* such that for each 3-tuple either it is not a tuple in Boats or there is a tuple in Reserves showing that sailor *I* has reserved it.

Find sailors who've reserved all boats (again!)

Simpler notation, same query. (Much clearer!)

To find sailors who've reserved all red boats:

Expressive Power of Algebra and Calculus

It is possible to write syntactically correct calculus queries that have an infinite number of answers! Such queries are called *unsafe*.

– e.g.,

It is known that every query that can be expressed in relational algebra can be expressed as a safe query in DRC / TRC; the converse is also true.

Relational Completeness: Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.

UNIT-III

Basic SQL Query

1.SQL Data definition

Introduction to Schema Refinement

Functional Dependencies

Normal Forms

Decompositions

Schema refinement in database design

Fourth Normal Form

Fifth normal form

The Form of a Basic SQL Queries:

History

IBM Sequel language developed as part of System R project at the IBM San Jose

Research Laboratory

Renamed Structured Query Language (SQL)

ANSI and ISO standard SQL:

– SQL-86

– SQL-89

– SQL-92

– SQL:1999 (language name became Y2K compliant!)

– SQL:2003

Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.

– Not all examples here may work on your particular system.

Data Definition Language

The schema for each relation, including attribute types.

Integrity constraints

Authorization information for each relation.

Non-standard SQL extensions also allow specification of

– The set of indices to be maintained for each relations.

– The physical storage structure of each relation on disk.

Create Table Construct

An SQL relation is defined using the **create table** command:

create table r (A_1 D_1, A_2 D_2, ..., A_n D_n,

(integrity-constraint$_1$),

...,

(integrity-constraint$_k$))

– r is the name of the relation

– each A_i is an attribute name in the schema of relation r

– D_i is the data type of attribute A_i

Example:

create table *branch*

(*branch_name* **char**(15)**,**

branch_city **char(**30),

assets **integer**)

Domain Types in SQL

char(n). Fixed length character string, with user-specified length *n.*

varchar(n). Variable length character strings, with user-specified maximum length *n.*

int. Integer (a finite subset of the integers that is machine-dependent).

smallint. Small integer (a machine-dependent subset of the integer domain type).

numeric(p,d). Fixed point number, with user-specified precision of *p* digits, with *n* digits to the right of decimal point.

real, double precision. Floating point and double-precision floating point numbers, with machine-dependent precision.

float(n). Floating point number, with user-specified precision of at least *n* digits.

More are covered in Chapter 4.

Integrity Constraints on Tables

not null

primary key (A_1, ..., A_n)

Basic Insertion and Deletion of Tuples

Newly created table is empty

Add a new tuple to *account*

insert into *account* **values** ('A-9732', 'Perryridge', 1200)

– Insertion fails if any integrity constraint is violated

Delete *all* tuples from *account* **delete**

from *account*

Drop and Alter Table Constructs

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

The **alter table** command is used to add attributes to an existing relation:

alter table *r* **add** *A D*

where *A* is the name of the attribute to be added to relation *r* and *D* is the domain of *A*.

– All tuples in the relation are assigned *null* as the value for the new attribute.

The **alter table** command can also be used to drop attributes of a relation:

alter table *r* **drop** *A*

where *A* is the name of an attribute of relation *r*

– Dropping of attributes not supported by many databases

Basic Query Structure

Atypical SQL query has the form:

select A_1, A_2, ...,
A_n **from** r_1, r_2, ...,
r_m **where** P

–A_i represents an attribute

–R_i represents a relation

–P is a predicate.

This query is equivalent to the relational algebra expression. The result of an SQL query is a relation.

The select Clause

The **select** clause list the attributes desired in the result of a query

–corresponds to the projection operation of the relational algebra

- Example: find the names of all branches in the *loan* relation:
  select *branch_name*
  from *loan*

In the relational algebra, the query would be:

Õ*branch_name* (*loan*)

NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

– E.g. *Branch_Name* ≡ *BRANCH_NAME* ≡ *branch_name*

– Some people use upper case wherever we use bold font.

SQL allows duplicates in relations as well as in query results.

To force the elimination of duplicates, insert the keyword **distinct** after select**.**

Find the names of all branches in the *loan* relations, and remove duplicates **select**

distinct *branch_name* **from** *loan*

The keyword **all** specifies that duplicates not be removed.

select all *branch_name* **from** *loan*

The select Clause (Cont.)

An asterisk in the select clause denotes "all attributes" **select**

* **from** *loan*

The **select** clause can contain arithmetic expressions involving the operation, +, –, *, and /, and operating on constants or attributes of tuples.

E.g.:

select *loan_number, branch_name, amount* * 100 **from** *loan*

The where Clause

The **where** clause specifies conditions that the result must satisfy

– Corresponds to the selection predicate of the relational algebra.

To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1200.

select *loan_number* **from** *loan* **where** *branch_name* = 'Perryridge' **and** *amount* 1200

Comparison results can be combined using the logical connectives **and, or,** and **not.**

The from Clause

The **from** clause lists the relations involved in the query

– Corresponds to the Cartesian product operation of the relational algebra.

Find the Cartesian product *borrower X loan*

Select *from *borrower, loan*

The Rename Operation

SQL allows renaming relations and attributes using the **as** clause:

old-name **as** *new-name*

E.g. Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

Tuple Variables

Tuple variables are defined in the **from** clause via the use of the **as** clause.

Find the customer names and their loan numbers and amount for all customers having a loan at some branch.

11.Example Basic Sql Queries:

Example Instances

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

R1

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

S1

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

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Slide No:L2-1

We will use these instances of the Sailors and Reserves relations in our examples.

If the key for the Reserves relation contained only the attributes *sid* and *bid*, how would the semantics differ?

Basic SQL Query

```
SELECT    [DISTINCT] target-list
FROM      relation-list
WHERE     qualification
```

relation-list A list of relation names (possibly with a *range-variable* after each name).

target-list A list of attributes of relations in *relation-list*

• *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of) combined using AND, OR and NOT.

DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated!

Conceptual Evaluation Strategy

Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

– Compute the cross-product of *relation-list*.

– Discard resulting tuples if they fail *qualifications*.

– Delete attributes that are not in *target-list*.

– If DISTINCT is specified, eliminate duplicate rows.

This strategy is probably the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND R.bid=103
```

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

Slide No:L2-4

A Note on Range Variables

Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103

SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
          AND bid=103
```

Find sailors who've reserved at least one boat

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid=R.sid
```

Would adding DISTINCT to this query make a difference?

What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to this variant of the query make a difference?

Expressions and Strings

Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters.*

AS and = are two ways to name fields in result.

LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.

String Operations

SQL includes a string-matching operator for comparisons on character strings. The operator "like" uses patterns that are described using two special characters:

– percent (%). The % character matches any substring.

– underscore (_). The _ character matches any character.

Find the names of all customers whose street includes the substring "Main".

select *customer_name*
from *customer*
where *customer_street* **like** '% Main%'

Match the name "Main%" **like** 'Main\%' **escape** '\'

SQL supports a variety of string operations such as

– concatenation (using "||")

– converting from upper to lower case (and vice versa)

– finding string length, extracting substrings, etc.

Ordering the Display of Tuples

List in alphabetic order the names of all customers having a loan in Perryridge branch

select distinct *customer_name*
from *borrower, loan*
where *borrower loan_number = loan.loan_number* **and**
branch_name = 'Perryridge'
order by *customer_name*

We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.

–Example: **order by** *customer_name* **desc**

Duplicates

In relations with duplicates, SQL can define how many copies of tuples appear in the result.

Multiset versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

σ_θ (r_1): If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ,, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.

2. **Π_A (r):** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.

r_1 x r_2 : If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are c_1 x c_2 copies of the tuple t_1. t_2 in r_1 x r_2

Example: Suppose multiset relations r_1 (A, B) and r_2 (C) are as follows:

r_1 = {(1, a) (2,a)} r_2 = {(2), (3), (3)} Then $\Pi_B(r_1)$ would be {(a), (a)}, while $\Pi_B(r_1)$ x r_2 would be

$\{(a,2), (a,2), (a,3), (a,3), (a,3), (a,3)\}$

SQL duplicate semantics:

select A_1,, A_2, ..., A_n
from r_1, r_2, ..., r_m
where P

is equivalent to the *multiset* version of the expression:

Nested Queries:

A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)

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

To find sailors who've *not* reserved #103, use NOT IN.

To understand semantics of nested queries, think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

Correlated Nested Queries:

Nested Queries with Correlation

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

EXISTS is another set comparison operator, like IN.

If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)

Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

Set comparison Operators:

Nested Sub queries

SQL provides a mechanism for the nesting of subqueries.

A **subquery** is a **select-from-where** expression that is nested within another query.

A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.

Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

Suppose a tuple occurs *m* times in *r* and *n* times in *s*, then, it occurs:

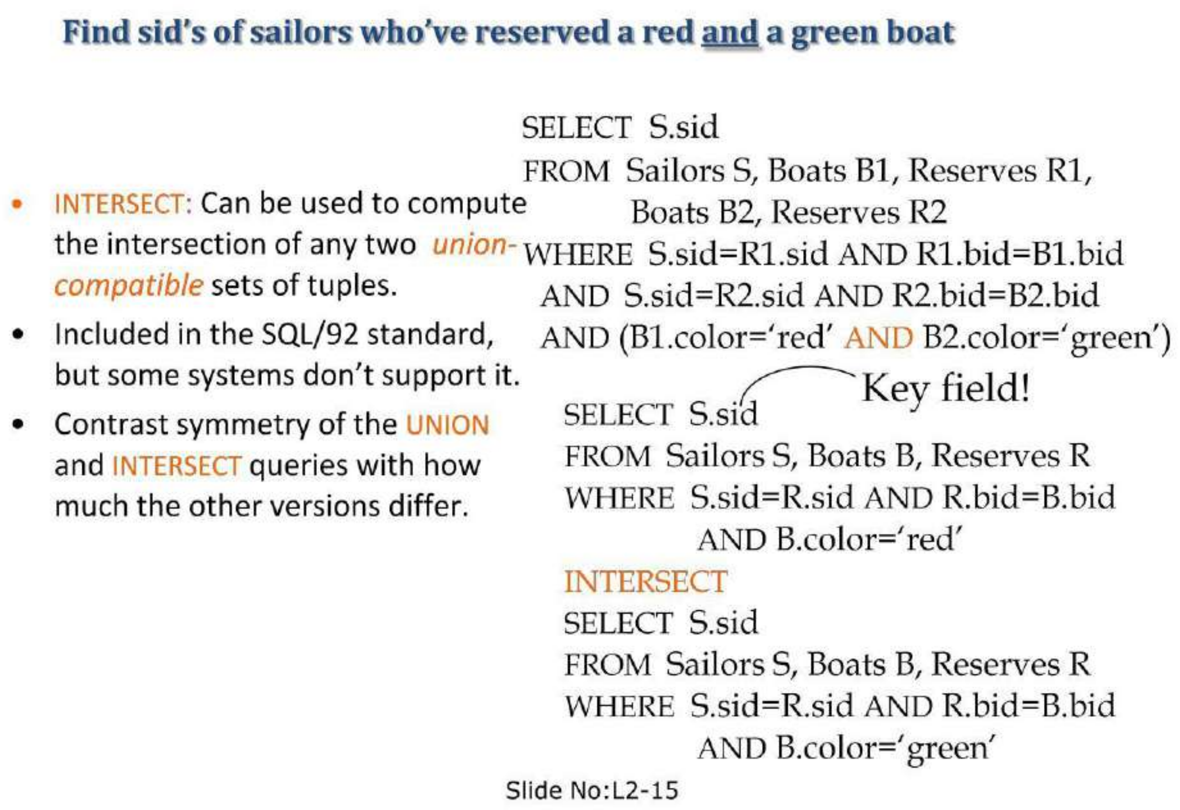
– $m + n$ times in *r* **union all** *s*

– min(m,n) times in r **intersect all** s

– max(0, $m - n$) times in r **except all** s

Find all customers who have a loan, an account, or both:

Find sid's of sailors who've reserved a red <u>or</u> a green boat



More on Set-Comparison Operators

We've already seen IN, EXISTS and UNIQUE. Can also use NOT IN, NOT EXISTS and NOT UNIQUE.

Also available: *op* ANY, *op* ALL, *op* IN

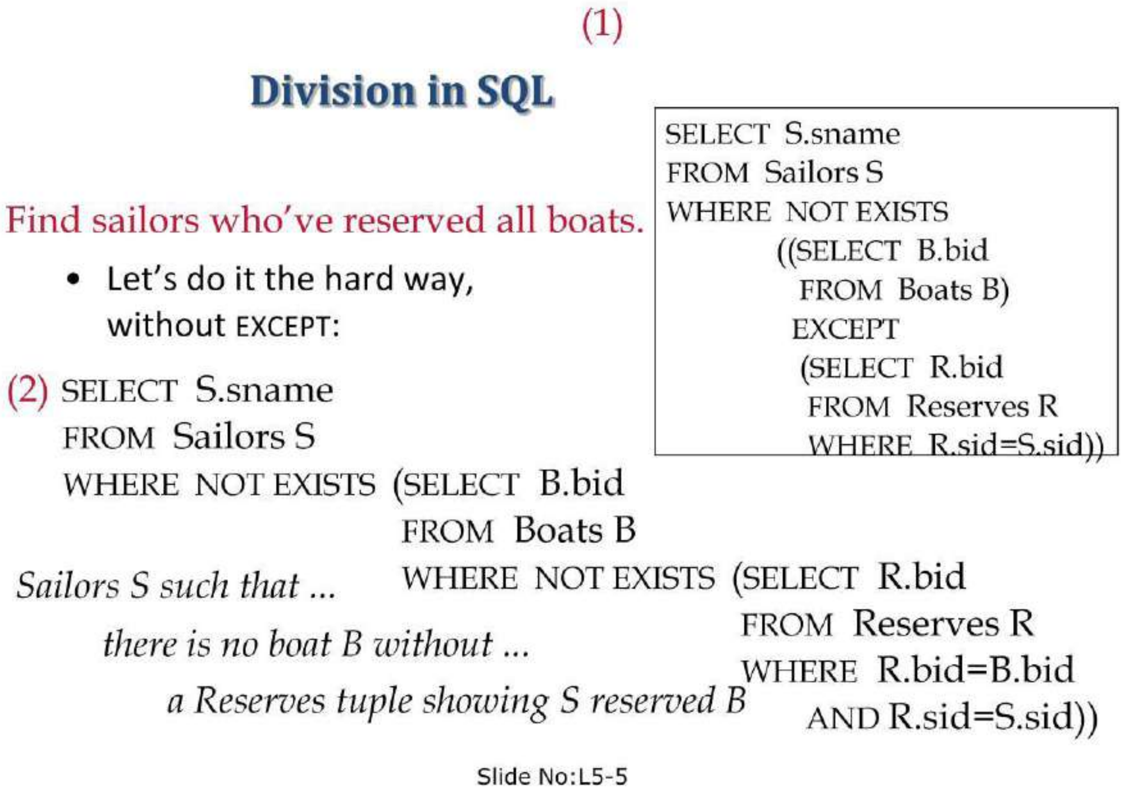
Find sailors whose rating is greater than that of some sailor called Horatio:

Rewriting INTERSECT Queries Using IN

Similarly, EXCEPT queries re-written using NOT IN.

To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in SELECT clause. (What about INTERSECT query?)

Division in SQL



Aggregate Operators:

These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value

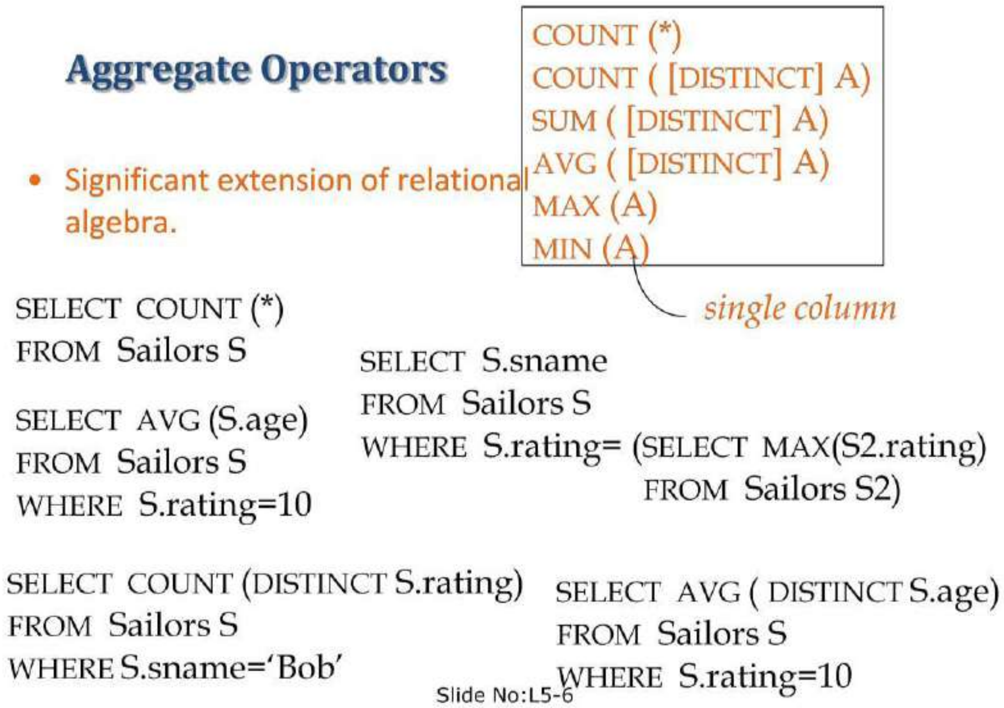
min: minimum value

max: maximum value

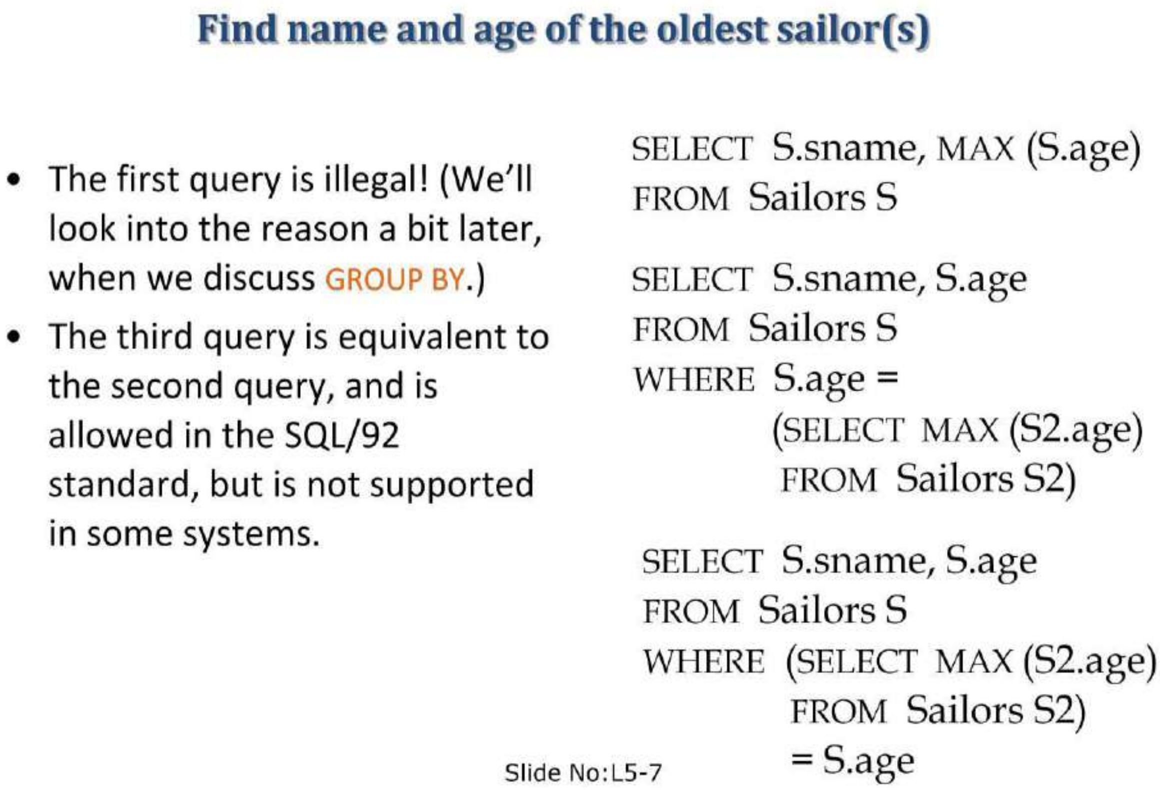
sum: sum of values

count: number of values

Aggregate Operators examples



Significant extension of relational algebra.



Motivation for Grouping

So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.

Consider: *Find the age of the youngest sailor for each rating level.*

– In general, we don't know how many rating levels exist, and what the rating values for these levels are!

– Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (!):

Queries With GROUP BY and HAVING

The *target-list* contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).

```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

– The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary'* fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.

The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a <u>*single value per group*</u>!

– In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)

One answer tuple is generated per qualifying group.

Find age of the youngest sailor with age 18, for each rating with at least 2 <u>such</u> sailors

Find age of the youngest sailor with age 18, for each rating with at least 2 <u>such</u> sailors

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

Sailors instance:

<u>sid</u>	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Slide No:L6-2

- Find age of the youngest sailor with age 18, for each rating with at least 2 <u>such</u> sailors and with every sailor under 60.

- Find age of the youngest sailor with age 18, for each rating with at least 2 sailors between 18 and 60.

For each red boat, find the number of reservations for this boat Grouping over a join of three relations.

What do we get if we remove *B.color='red'* from the WHERE clause and add a HAVING clause with this condition?

What if we drop Sailors and the condition involving S.sid?

Find age of the youngest sailor with age > 18, for each rating with at least 2 sailors (of any age)

Shows HAVING clause can also contain a subquery.

Compare this with the query where we considered only ratings with 2 sailors over 18!

What if HAVING clause is replaced by:

– HAVING COUNT(*) >1

Find those ratings for which the average age is the minimum over all ratings

Aggregate operations cannot be nested! WRONG:

Find the average account balance at the Perryridge branch.

Aggregate Functions – Group By

Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
    from depositor, account
    where depositor.account_number = account.account_number
    group by branch_name
```

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having Clause

Find the names of all branches where the average account balance is more than $1,200.

select *branch_name,* **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values:

Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable* (e.g., no spouse's name).

– SQL provides a special value *null* for such situations.

The presence of *null* complicates many issues. E.g.:

– Special operators needed to check if value is/is not *null*.

– Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?

– We need a 3-valued logic (true, false and *unknown*).

– Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)

– New operators (in particular, *outer joins*) possible/needed.

Comparision Using 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 predicate **is null** can be used to check for null values.

– Example: Find all loan number which appear in the *loan* relation with null values for *amount.*

select *loan_number*
from *loan*
where *amount* **is null**

The result of any arithmetic expression involving *null* is *null*

– Example: 5 + *null* returns null

However, aggregate functions simply ignore nulls

– More on next slide

Null Values and Three Valued Logic

Any comparison with *null* returns *unknown*

– Example*: 5 < null or null <> null or null = null*

Logical Connectives:AND,OR,NOT

Three-valued logic using the truth value *unknown*:

– 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*

– "*P* **is unknown**" evaluates to true if predicate *P* evaluates to *unknown*

Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*

Null Values and Aggregates

Total all loan amounts

select sum (*amount*)

from *loan*

– Above statement ignores null amounts

– Result is *null* if there is no non-null amount

All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Impact on SQL Constructs:

"In" Construct

- Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                       from depositor )
```

- Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                           from depositor )
```

Slide No:L3-8

Example Query

- Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
         branch_name = 'Perryridge'  and
         (branch_name, customer_name ) in
                (select branch_name, customer_name
                 from depositor, account
                 where depositor.account_number =
                     account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Slide No:L3-9

"Some" Construct

"Some" Construct

- Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
               S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
              (select assets
               from branch
               where branch_city = 'Brooklyn')
```

Slide No:L4-1

“All” Construct

Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
      from branch
      where assets > all
            (select assets
            from branch
            where branch_city = 'Brooklyn')
```

“Exists” Construct

Find all customers who have an account at all branches located in Brooklyn.

Absence of Duplicate Tuples

The **unique** construct tests whether a subquery has any duplicate tuples in its result.

Find all customers who have at most one account at the Perryridge branch.

```
select distinct S.customer_name
      from depositor as S
      where not exists (
            (select branch_name
            from branch
            where branch_city = 'Brooklyn')
            except
            (select R.branch_name
            from depositor as T, account as R
            where T.account_number = R.account_number and
                  S.customer_name = T.customer_name ))
```

select *T.customer_name*

from *depositor* **as** *T*

where unique (

select *R.customer_name*

from *account, depositor* **as** *R*

where *T.customer_name* = *R.customer_name* **and**

R.account_number = *account.account_number* **and**

account.branch_name = 'Perryridge')

Example Query

Find all customers who have at least two accounts at the Perryridge branch.

select distinct *T.customer_name*
from *depositor* **as** *T*
where not unique (
select *R.customer_name*
from *account, depositor* **as** *R*
where *T.customer_name* = *R.customer_name* **and**
R.account_number = *account.account_number* **and**
account.branch_name = 'Perryridge')

Modification of the Database – Deletion

Delete all account tuples at the Perryridge branch

delete from *account*

where *branch_name* = 'Perryridge'

Delete all accounts at every branch located in the city 'Needham'.

delete from *account*

where *branch_name* **in** (**select** *branch_name*

from *branch*

where *branch_city* = 'Needham')

Example Query

Delete the record of all accounts with balances below the average at the bank.

Modification of the Database – Insertion

Add a new tuple to *account*

insert into *account*
values ('A-9732', 'Perryridge', 1200) or equivalently
insert into *account* (*branch_name, balance, account_number*)
values ('Perryridge', 1200, 'A-9732')

Add a new tuple to *account* with *balance* set to null

insert into *account* **values** ('A-777','Perryridge', *null*)

Modification of the Database – Insertion

Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account

insert into *account*
select *loan_number, branch_name*, 200
from *loan*
where *branch_name* = 'Perryridge'

insert into *depositor*
select *customer_name, loan_number*
from *loan, borrower*
where branch_name = 'Perryridge'
and *loan.account_number* = *borrower.account_number*

The **select from where** statement is evaluated fully before any of its results are inserted into the relation

–Motivation: **insert into** *table*1 **select** * **from** *table*1

Modification of the Database – Updates

Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

– Write two **update** statements:

update *account*
set *balance* = *balance* ∗ 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* ∗ 1.05
where *balance* ≤ 10000

– The order is important

– Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

update *account*
set *balance* = **case**
when *balance* <= 10000 **then** *balance* *1.05
else *balance* * 1.06
end

20. Outer Joins:

Joined Relations**

Join operations take two relations and return as a result another relation.

These additional operations are typically used as subquery expressions in the **from** clause

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.

Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Joined Relations – Datasets for Examples

Relation *loan*

loan_number	branch_name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	loan_number
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

Joined Relations – Examples

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

loan **inner join** *borrower* **on** *loan.loan_number = borrower.loan_number*

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	*null*	*null*

Joined Relations – Examples

loan **natural inner join** *borrower*

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	*null*	*null*	Hayes

Joined Relations – Examples

loan_number	*branch_name*	*amount*	*customer_name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Natural join can get into trouble if two relations have an attribute with same name that should not affect the join condition

– e.g. an attribute such as *remarks* may be present in many tables

Solution:

– *loan* **full outer join** *borrower* **using** (*loan_number*)

Derived Relations

SQL allows a subquery expression to be used in the **from** clause

Find the average account balance of those branches where the average account balance is greater than $1200.

select *branch_name, avg_balance*

from (**select** *branch_name,* **avg** (*balance*)

from *account*

group by *branch_name*) **as** *branch_avg* (*branch_name, avg_balance*)

where *avg_balance* > 1200

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

Complex Integrity Constraints in SQL:

Integrity Constraints (Review)

An IC describes conditions that every *legal instance* of a relation must satisfy.

– Inserts/deletes/updates that violate IC's are disallowed.

– Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)

Types of IC's: Domain constraints, primary key constraints, foreign key constraints, general constraints.

– *Domain constraints*: Field values must be of right type. Always enforced.

General Constraints

```
CREATE TABLE Reserves
        ( sname CHAR(10),
        bid INTEGER,
        day DATE,
        PRIMARY KEY (bid,day),
        CONSTRAINT noInterlakeRes
        CHECK (`Interlake' <>
                    ( SELECT B.bname
                    FROM Boats B
                    WHERE B.bid=bid)))
```

Useful when more general ICs than keys are involved.

Can use queries to express constraint.

Constraints can be named.

Constraints Over Multiple Relations

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- ASSERTION is the right solution; not associated with either table.

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

Number of boats plus number of sailors is < 100

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

Slide No:L8-8

Triggers and Active Databases:

Trigger: procedure that starts automatically if specified changes occur to the DBMS

Three parts:

– Event (activates the trigger)

– Condition (tests whether the triggers should run)

– Action (what happens if the trigger runs)

Triggers: Example (SQL:1999)

CREATE TRIGGER youngSailorUpdate

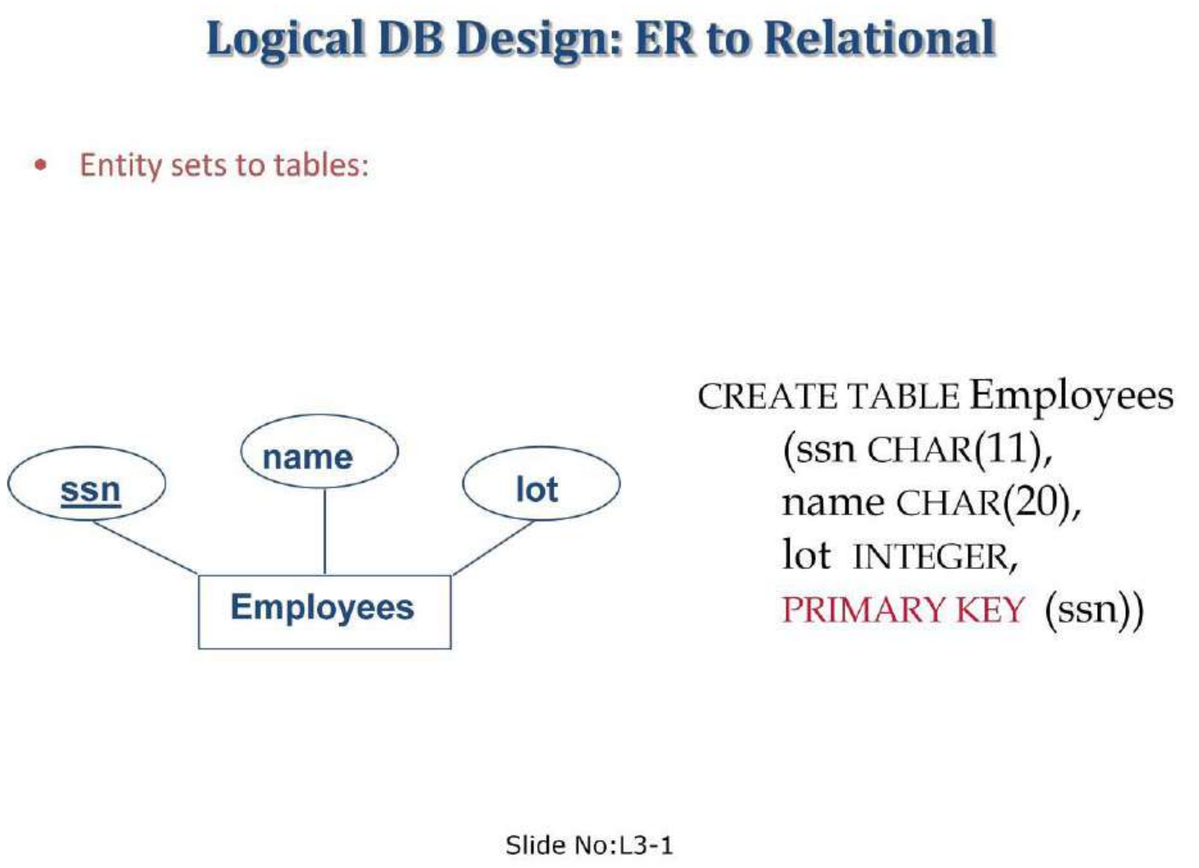
AFTER INSERT ON SAILORS

REFERENCING NEW TABLE NewSailors

FOR EACH STATEMENT

```
INSERT
INTO YoungSailors(sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors N
WHERE N.age <= 18
```

Logical DB Design:



Entity sets to tables:

Relationship Sets to Tables

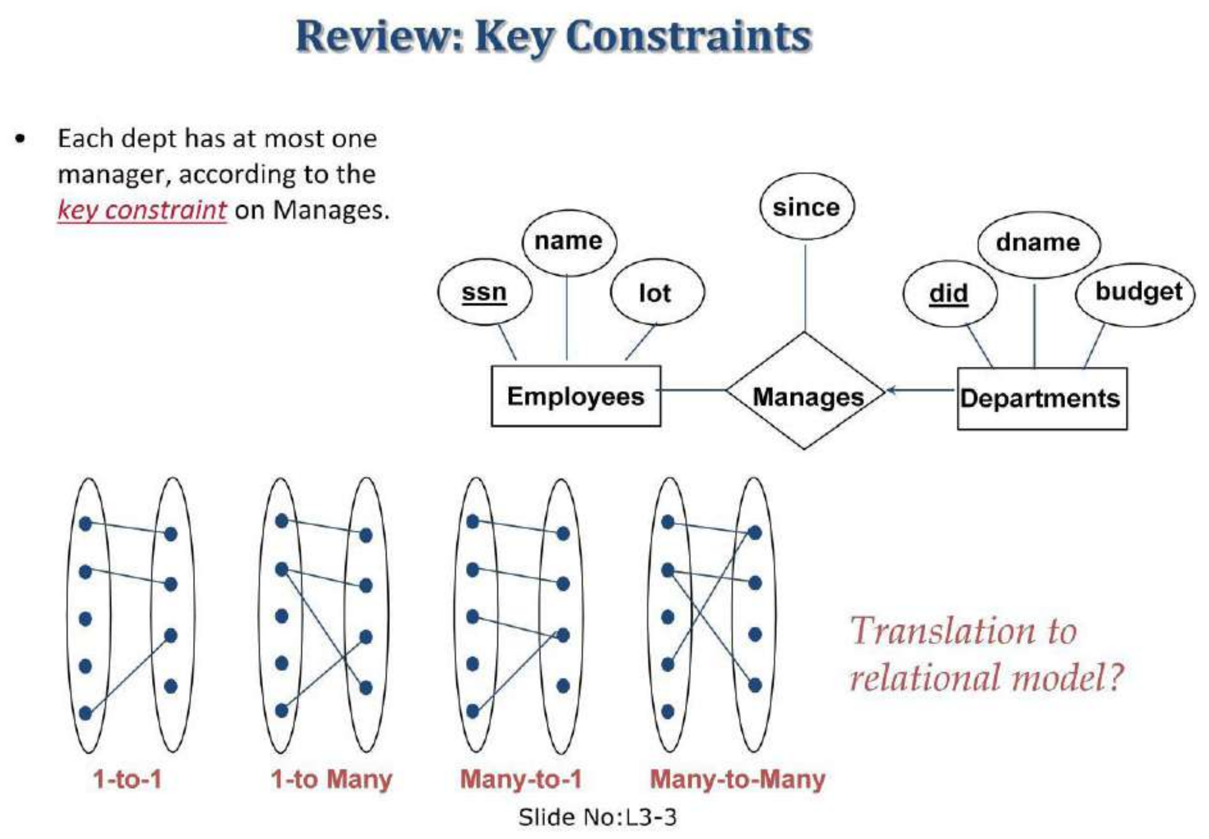
In translating a relationship set to a relation, attributes of the relation must include:

–Keys for each participating entity set (as foreign keys).

This set of attributes forms a *superkey* for the relation.

–All descriptive attributes.

Review: Key Constraints



- Each dept has at most one manager, according to the <u>*key constraint*</u> on Manages.

Translating ER Diagrams with Key Constraints

Map relationship to a table:

– Note that did is the key now!

– Separate tables for Employees and Departments.

Since each department has a unique manager, we could instead combine Manages and Departments.

Views and Security

Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).

–Given YoungStudents, but not Students or Enrolled, we can find students s who have are enrolled, but not the *cid's* of the courses they are enrolled in.

View Definition

A relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

A view is defined using the **create view** statement which has the form

create view *v* **as** < query expression >

where <query expression> is any legal SQL expression. The view name is represented by *v*.

Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

Example Queries

A view consisting of branches and their customers

Uses of Views

Hiding some information from some users

–Consider a user who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount.

–Define a view

(**create view** *cust_loan_data* **as**
select *customer_name, borrower.loan_number, branch_name* **from** *borrower, loan*
where *borrower.loan_number = loan.loan_number*)

–Grant the user permission to read *cust_loan_data*, but not *borrower or loan*

Predefined queries to make writing of other queries easier

–Common example: Aggregate queries used for statistical analysis of data

–Processing of Views

When a view is created

– the query expression is stored in the database along with the view name

–the expression is substituted into any query using the view

Views definitions containing views

–One view may be used in the expression defining another view

–A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1

– A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2

–A view relation v is said to be *recursive* if it depends on itself.

View Expansion

A way to define the meaning of views defined in terms of other views.

Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.

View expansion of an expression repeats the following replacement step:

repeat
Find any view relation v_i in e_1
Replace the view relation v_i by the expression defining
v_i **until** no more view relations are present in e_1

As long as the view definitions are not recursive, this loop will terminate

With Clause

The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

Find all accounts with the maximum balance

with *max_balance* (*value*) **as**

select max (*balance*)

from *account*
select *account_number*

from *account, max_balance*

where *account.balance = max_balance.value*

Complex Queries using With Clause

Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

Update of a View

Create a view of all loan data in the *loan* relation, hiding the *amount* attribute

create view *loan_branch* **as**

select *loan_number, branch_name*

from *loan*

Add a new tuple to *loan_branch*

insert into *loan_branch*

values ('L-37‘, 'Perryridge‘)

This insertion must be represented by the insertion of the tuple

('L-37', 'Perryridge', *null*) into the *loan* relation

Destroying and Altering Tables and Views:

Destroys the relation Students. The schema information *and* the tuples are deleted.

Adding and Deleting Tuples

Can insert a single tuple using:

What if Policies is a weak entity set?

Views

A *view* is just a relation, but we store a *definition*, rather than a set of tuples.

Introduction To Schema Refinement:

The Evils of Redundancy

Redundancy is at the root of several problems associated with relational schemas:

–redundant storage, insert/delete/update anomalies

Integrity constraints, in particular *functional dependencies*, can be used to identify schemas with such problems and to suggest refinements.

Main refinement technique: ***decomposition*** (replacing ABCD with, say, AB and BCD, or ACD and ABD).

Decomposition should be used judiciously:

–Is there reason to decompose a relation?

–What problems (if any) does the decomposition cause?

Problems Caused by Redundancy:

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

Redundant storage: Some information is stored repeatedly.

Update anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.

Insertion anomalies: It may not be possible to store some information unless some other information is stored as well.

Deletion anomalies: It may not be possible to delete some information without losing some other information as well.

Consider a relation obtained by translating a variant of the Hourly Emps entity set

Ex: Hourly Emps(*ssn*, *name*, *lot*, *rating*, *hourly wages*, *hours worked*)

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

Decompositions:

Intuitively, redundancy arises when a relational schema forces an association between attributes that is not natural.

Functional dependencies (ICs) can be used to identify such situations and to suggest revetments to the schema.

The essential idea is that many problems arising from redundancy can be addressed by

rating	***hourly wages***
8	10
5	7

ssn	***name***	***lot***	***rating***	***hours worked***
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

replacing a relation with a collection of smaller relations.

Each of the smaller relations contains a subset of the attributes of the original relation.

We refer to this process as decomposition of the larger relation into the smaller relations

We can deal with the redundancy in Hourly Emps by decomposing it into two relations:

Hourly Emps2(*ssn*, *name*, *lot*, *rating*, *hours worked*)

Wages(*rating*, *hourly wages*)

Problems Related to Decomposition:

Unless we are careful, decomposing a relation schema can create more problems than it solves.

Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?

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

To help with the rst question, several *normal forms* have been proposed for relations.

If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the n

Functional Dependencies (FDs):

A <u>functional dependency</u> XY holds over relation R if, for every allowable instance *r* of R:

– *t1* *r*, *t2* *r*, (*t1*) = (*t2*) implies (*t1*) = (*t2*)

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

An FD is a statement about *all* allowable relations.

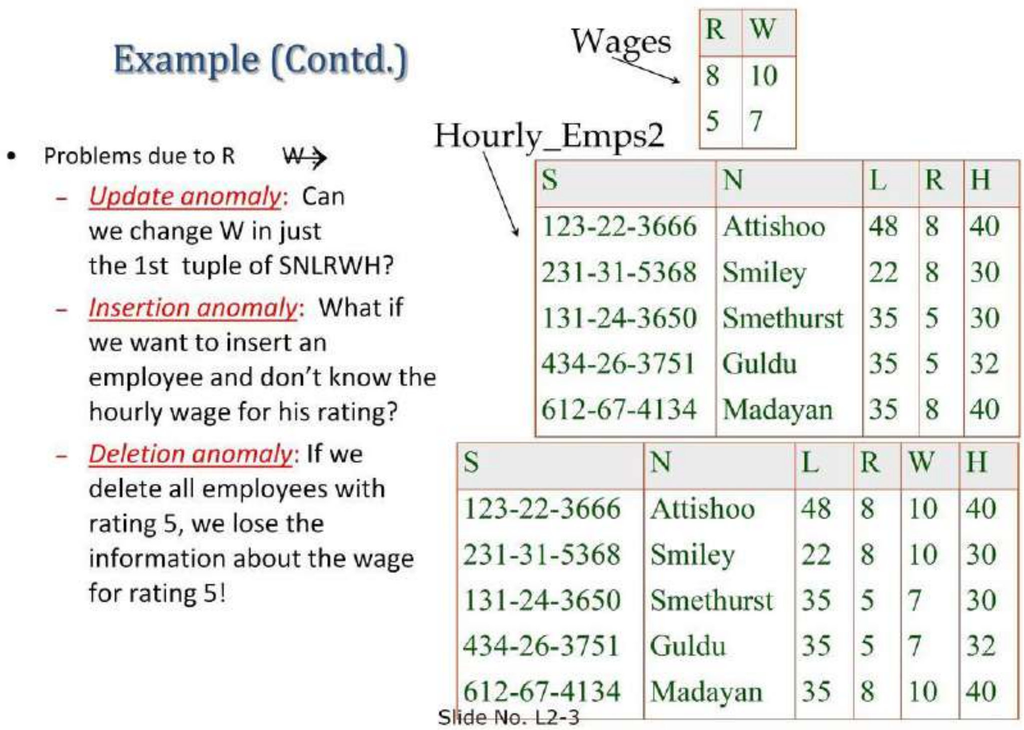
–Must be identified based on semantics of application.

– Given some allowable instance *r1* of R, we can check if it violates some FD *f*, but we cannot tell if *f* holds over R!

• K is a candidate key for R means that K R

– However, K R does not require K to be *minimal*!

Example: Constraints on Entity Set



Consider relation obtained from Hourly_Emps:

–Hourly_Emps (<u>*ssn*</u>, *name*, *lot*, *rating*, *hrly_wages*, *hrs_worked*)

<u>*Notation*</u>: We will denote this relation schema by listing the attributes: SNLRWH

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

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

Some FDs on Hourly_Emps:

– *ssn* is the key: S SNLRWH

– *rating* determines *hrly_wages*: R W

Constraints on a Relationship Set:

Suppose that we have entity sets Parts, Suppliers, and Departments, as well as a relationship set Contracts that involves all of them. We refer to the schema for Contracts as *CQPSD*. A contract with contract id *C* species that a supplier *S* will supply some quantity *Q* of a part *P* to a department *D*.

We might have a policy that a department purchases at most one part from any given supplier.

Thus, if there are several contracts between the same supplier and department, we know that the same part must be involved in all of them. This constraint is an FD, *DS ! P*.

Reasoning about FDs

Given some FDs, we can usually infer additional FDs:

– *ssn* *did*, *did* *lot* implies *ssn* *lot*

An FD *f* is <u>*implied by*</u> a set of FDs *F* if *f* holds whenever all FDs in *F* hold.

– = *closure of F* is the set of all FDs that are implied by *F*.

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

– <u>*Reflexivity*</u>: If X → Y, then Y → X

– <u>*Augmentation*</u>: If X → Y, then XZ → YZ for any Z

– <u>*Transitivity*</u>: If X → Y and Y →Z, then X →Z

These are *sound* and *complete* inference rules for FDs!

Couple of additional rules (that follow from AA):

– *Union*: If X → Y and X → Z, then X → YZ

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

Example: Contracts(*cid,sid,jid,did,pid,qty,value*), and:

– C is the key: C →CSJDPQV

– Project purchases each part using single contract:

– JP → C

– Dept purchases at most one part from a supplier: S

– D → P

• JP → C, C → CSJDPQV imply JP →CSJDPQV

• SD → P implies SDJ → JP

• SDJ → JP, JP → CSJDPQV imply SDJ → CSJDPQV

Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)

• Typically, we just want to check if a given FD *X* *Y* is in the closure of a set of FDs *F*. An efficient check:

– Compute <u>*attribute closure*</u> of X (denoted) wrt *F:*

• Set of all attributes A such that X A is in

• There is a linear time algorithm to compute this.

– Check if Y is in

• Does F = {A → B, B →C, C D → E } imply A → E?

– i.e, is A → E in the closure ? Equivalently, is E in ?

Closure of a Set of FDs

The set of all FDs implied by a given set *F* of FDs is called the **closure of F** and is denoted as *F+*.

An important question is how we can **infer**, or compute, the closure of a given set *F* of FDs.

The following three rules, called **Armstrong's Axioms**, can be applied repeatedly to infer all FDs implied by a set *F* of FDs.

We use *X*, *Y*, and *Z* to denote *sets* of attributes over a relation schema *R*:

Reflexivity: If *X Y*, then *X !Y*.

Augmentation: If *X ! Y*, then *XZ ! YZ* for any *Z*.

Transitivity: If *X ! Y* and *Y ! Z*, then *X ! Z*.

Armstrong's Axioms are **sound** in that they generate only FDs in *F+* when applied to a set *F* of FDs.

They are **complete** in that repeated application of these rules will generate all FDs in the closure *F+*.

It is convenient to use some additional rules while reasoning about *F+*:

Union: If *X ! Y* and *X ! Z*, then *X !YZ*.

Decomposition: If *X ! YZ*, then *X !Y* and *X ! Z*.

These additional rules are not essential; their soundness can be proved using Armstrong's Axioms.

Attribute Closure

If we just want to check whether a given dependency, say, $X \rightarrow Y$, is in the closure of a set F of FDs,

we can do so eciently without computing $F+$. We rst compute the **attribute closure** $X+$ with respect to F,

which is the set of attributes A such that $X \rightarrow A$ can be inferred using the Armstrong Axioms.

The algorithm for computing the attribute closure of a set X of attributes is

closure = X;

repeat until there is no change: {

if there is an FD $U \rightarrow V$ in F such that U *subset of closure*,

then set *closure* = *closure union of* V}

Normal Forms:

The normal forms based on FDs are *rst normal form (1NF)*, *second normal form (2NF)*, *third normal form (3NF)*, and *Boyce-Codd normal form (BCNF)*.

These forms have increasingly restrictive requirements: Every relation in BCNF is also in 3NF,

every relation in 3NF is also in 2NF, and every relation in 2NF is in 1NF.

A relation

is in **first normal form** if every field contains only atomic values, that is, not lists or sets.

This requirement is implicit in our defition of the relational model.

Although some of the newer database systems are relaxing this requirement

2NF is mainly of historical interest.

3NF and BCNF are important from a database design standpoint.

Normal Forms

Returning to the issue of schema refinement, the first question to ask is whether any

refinement is needed!

If a relation is in a certain *normal form* (BCNF, 3NF etc.), it is known that certain kinds

of problems are avoided/minimized. This can be used to help us decide whether decomposing the relation will help

Role of FDs in detecting redundancy:

– Consider a relation R with 3 attributes, ABC.

No FDs hold: There is no redundancy here.

• Given A,B: Several tuples could have the same A value, and if so, they'll all have the same B value!

First Normal Form:

1NF (First Normal Form)

a relation R is in 1NF if and only if it has only single-valued attributes (atomic values)

EMP_PROJ (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME, PNAME, PLOCATION)

solution: decompose the relation

EMP_PROJ2 (<u>SSN</u>, <u>PNO</u>, HOURS, ENAME,

PNAME) LOC (<u>PNO</u>, <u>PLOCATION</u>)

Second Normal Form:

2NF (Second Normal Form)

a relation R in 2NF if and only if it is in 1NF and every nonkey column depends on a key not a subset of a key

all nonprime attributes of R must be fully functionally dependent on a whole key(s) of the relation, not a part of the key

no violation: single-attribute key or no nonprime attribute
2NF (Second Normal Form)

- violation: part of a key → nonkey

EMP_PROJ2 (SSN, PNO, HOURS, ENAME, PNAME)

SSN → ENAME

PNO → PNAME

- solution: decompose the relation

EMP_PROJ3 (SSN, PNO, HOURS)

EMP (SSN, ENAME)

PROJ (PNO, PNAME)

Third Normal Form:

3NF (Third Normal Form)

a relation R in 3NF if and only if it is in 2NF and every nonkey column does not depend on another nonkey column

- all nonprime attributes of R must be non-transitively functionally dependent on a key of the relation

violation: nonkey → nonkey
3NF (Third Normal Form)

- SUPPLIER (SNAME, STREET, CITY, STATE, TAX)

SNAME → STREET, CITY, STATE

STATE $\rightarrow$ TAX (nonkey $\rightarrow$ nonkey)

SNAME $\rightarrow$ STATE $\rightarrow$ TAX (transitive FD)

- solution: decompose the relation

SUPPLIER2 (<u>SNAME</u>, STREET, CITY, STATE)

TAXINFO (<u>STATE</u>, TAX)

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs *F* is in BCNF if, for all X A in

– A $\rightarrow$ X (called a *trivial* FD), or

– X contains a key for R.

In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.

–No dependency in R that can be predicted using FDs alone.

– If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.

–If example relation is in BCNF, the 2 tuples must be identical(since X is a key).

Third Normal Form (3NF)

- Relation R with FDs *F* is in 3NF if, for all X $\rightarrow$ A in

–A $\rightarrow$X (called a *trivial* FD), or

–X contains a key for R, or

–A is part of some key for R.

Minimality of a key is crucial in third condition above!

If R is in BCNF, obviously in 3NF.

BCNF:

If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no ``good'' decomp, or performance considerations).

–Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.

Properties of Decompositions :

Suppose that relation R contains attributes *A1 ... An.* A *<u>decomposition</u>* of R consists of replacing R by two or more relations such that:

– Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and

–Every attribute of R appears as an attribute of one of the new relations.

Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.

E.g., Can decompose SNLRWH into SNLRH and RW.

Example Decomposition

Decompositions should be used only when needed.

– SNLRWH has FDs S SNLRWH and R W

– Second FD causes violation of 3NF; W values repeatedly associated with R values. Easiest way to fix this is to create a relation RW to store these associations , and to remove W from the main schema:

i.e., we decompose SNLRWH into SNLRH and RW

The information to be stored consists of SNLRWH tuples. If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

There are three potential problems to consider:

–Some queries become more expensive.

e.g., How much did sailor Joe earn? (salary = W*H)

– Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation!

Fortunately, not in the SNLRWH example.

– Checking some dependencies may require joining the instances of the decomposed relations.

Fortunately, not in the SNLRWH example.

Tradeoff: Must consider these issues vs. redundancy.

Lossless Join Decompositions:

Decomposition of R into X and Y is *lossless-join* w.r.t. a set of FDs F if, for every

instance *r* that satisfies F:

– (r) (r) = r

- It is always true that r (r) (r)

– In general, the other direction does not hold! If it does, the decomposition is lossless-join.

Definition extended to decomposition into 3 or more relations in a straightforward way.

It is essential that all decompositions used to deal with redundancy be lossless! (Avoids Problem (2).)

More on Lossless Join

- The decomposition of R into X and Y is lossless-join wrt F if and only if the closure of F contains:
  - X Y X, or
  - X Y Y
- In particular, the decomposition of R into UV and R - V is lossless-join if U V holds over R.

A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

$\rightarrow$

Slide No. L4-5

Dependency Preserving Decomposition

- Consider CSJDPQV, C is key, JP C and SD P.

–BCNF decomposition: CSJDQV and SDP

– Problem: Checking JP C requires a join!

Dependency preserving decomposition (Intuitive):

– If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. *(Avoids Problem (3).)*

Projection of set of FDs F: If R is decomposed into X, ... projection of F onto X

enoted F_X) is the set of FDs U V in F^+ (*closure of F*) such that U, V are in X.

Decomposition of R into X and Y is *dependency preserving*

if $(F_X$ union $F_Y)^+ = F^+$

– i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.

Important to consider F^+, not F, in this definition:

– ABC, A B, B C, C A, decomposed into AB and BC.

– Is this dependency preserving? Is C A preserved?????

Dependency preserving does not imply lossless join:

–ABC, AB, decomposed into AB and BC.

And vice-versa! (Example?)

Decomposition into BCNF

Consider relation R with FDs F. If X Y violates BCNF, decompose R into R - Y and XY.

– Repeated application of this idea will give us a collection of relations that are in BCNF; lossless join decomposition, and guaranteed to terminate.

– e.g., CSJDPQV, key C, JP C, SD P, J S – To deal with SD P,

decompose into SDP, CSJDQV.

–To deal with JS, decompose CSJDQV into JS and CJDQV

In general, several dependencies may cause violation of BCNF. The order in which we ``deal with'' them could lead to very different sets of relations!

BCNF and Dependency Preservation

In general, there may not be a dependency preserving decomposition into BCNF.

– e.g., CSZ, CS Z, Z C

–Can't decompose while preserving 1st FD; not in BCNF.

Similarly, decomposition of CSJDQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs JP C, SD P and J S).

–However, it is a lossless join decomposition.

– In this case, adding JPC to the collection of relations gives us a dependency preserving decomposition.

JPC tuples stored only for checking FD! (*Redundancy!*)

Decomposition into 3NF

Obviously, the algorithm for lossless join decomp into BCNF can be used to obtain a lossless join decomp into 3NF (typically, can stop earlier).

To ensure dependency preservation, one idea:

–If XY is not preserved, add relation XY.

– Problem is that XY may violate 3NF! e.g., consider the addition of CJP to `preserve' JP C. What if we also have J C ?

Refinement: Instead of the given set of FDs F, use a *minimal cover for F*.

Schema Refinement in Data base Design:

Constraints on an Entity Set

Consider the Hourly Emps relation again. The constraint that attribute *ssn* is a key can be expressed as an FD:

{ ssn }-> { ssn, name, lot, rating, hourly wages, hours worked}

For brevity, we will write this FD as *S -> SNLRWH*, using a single letter to denote each attribute

In addition, the constraint that the *hourly wages* attribute is determined by the *rating* attribute is an

FD: $R \rightarrow W$.

Constraints on a Relationship Set

The previous example illustrated how FDs can help to rene the subjective decisions made during ER design,

but one could argue that the best possible ER diagram would have led to the same nal set of relations.

Our next example shows how FD information can lead to a set of relations that eliminates some redundancy problems and is unlikely to be arrived at solely through ER design.

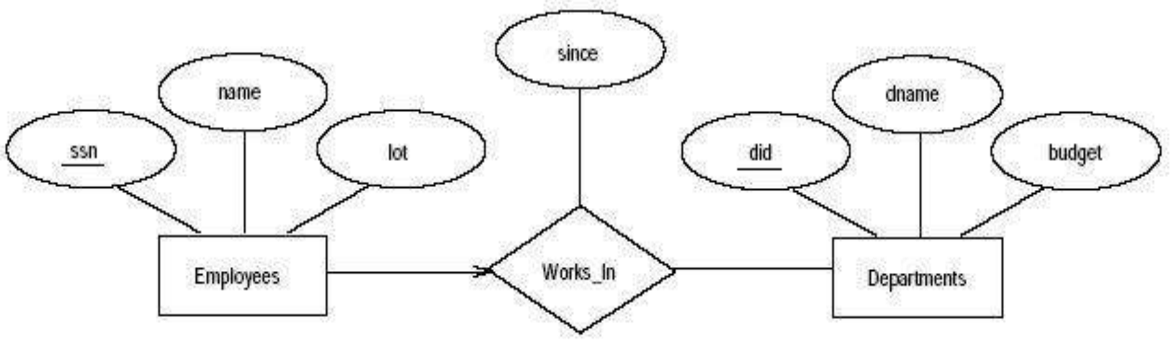
Identifying Attributes of Entities

in particular, it shows that attributes can easily be associated with the `wrong' entity set during ER design.

The ER diagram shows a relationship set called Works In that is similar to the Works In relationship set

Using the key constraint, we can translate this ER diagram into two relations:

Workers(*ssn*, *name*, *lot*, *did*, *since*)



Identifying Entity Sets

Let Reserves contain attributes *S*, *B*, and *D* as before, indicating that sailor *S* has a reservation for boat *B* on day *D*.

In addition, let there be an attribute *C* denoting the credit card to which the reservation is charged.

Suppose that every sailor uses a unique credit card for reservations. This constraint is expressed by the FD *S* -> *C*. This constraint indicates that in relation Reserves, we store the credit card number

for a sailor as often as we have reservations for that

sailor, and we have redundancy and potential update anomalies.

Multivalued Dependencies:

Suppose that we have a relation with attributes *course*, *teacher*, and *book*, which we denote as *CTB*.

The meaning of a tuple is that teacher *T* can teach course *C*, and book *B* is a recommended text for the course.

There are no FDs; the key is *CTB*. However, the recommended texts for a course are independent of the instructor.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

There are three points to note here:

The relation schema *CTB* is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over *CTB*.

There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.

The redundancy can be eliminated by decomposing *CTB* into *CT* and *CB*.

Let R be a relation schema and let X and Y be subsets of the attributes of R. Intuitively,

the **multivalued dependency** *X !! Y* is said to hold over R if, in every legal

The redundancy in this example is due to the constraint that the texts for a course are independent of the instructors, which cannot be epressed in terms of FDs.

This constraint is an example of a *multivalued dependency*, or MVD. Ideally, we should model this situation using two binary relationship sets, Instructors with attributes *CT* and Text with attributes *CB*.

Because these are two essentially independent relationships, modeling them with a single ternary relationship set with attributes *CTB* is inappropriate.

Three of the additional rules involve only MVDs:

MVD Complementation: If $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow R - XY$

MVD Augmentation: If $X \rightarrow\rightarrow Y$ and $W > Z$, then

$WX \rightarrow\rightarrow YZ$.

MVD Transitivity: If $X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z$, then

$X \rightarrow\rightarrow (Z - Y)$.

Fourth Normal Form:

R is said to be in **fourth normal form (4NF)** if for every MVD $X \rightarrow\rightarrow Y$ that holds over

R, one of the following statements is true:

Y *subset of* X or $XY = R$, or

X is a superkey.

Join Dependencies:

A join dependency is a further generalization of MVDs. A **join dependency** (JD) $\infty\{$

$R1, \ldots.\ Rn\ \}$ is said to hold over a relation R if $R1, \ldots.\ Rn$ is a lossless-join decomposition of R.

An MVD X ->-> Y over a relation R can be expressed as the join dependency ∞ {

XY,X(R−Y)}

As an example, in the *CTB* relation, the MVD C ->->T can be expressed as the join

dependency ∞\{ CT, CB\}

Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

Fifth Normal Form:

A relation schema R is said to be in **fth normal form (5NF)** if for every JD $\infty\{\ R1, \ldots.$

$Ri = R$ for some i, or

The JD is implied by the set of those FDs over R in which the left side is a key for R.

The following result, also due to Date and Fagin, identies conditions|again, detected using

only FD information|under which we can safely ignore JD information.

If a relation schema is in 3NF and each of its keys consists of a single attribute,it is also in

5NF.

Inclusion Dependencies:

MVDs and JDs can be used to guide database design, as we have seen, although they are less common than FDs and harder to recognize and reason about.

In contrast, inclusion dependencies are very intuitive and quite common. However, they typically have little influence on database design

The main point to bear in mind is that we should not split groups of attributes that participate in an inclusion dependency.

Most inclusion dependencies in practice are *key-based*, that is, involve only keys.

UNIT-IV

Transaction Management

- ACID Properties
- Need for concurrency control
- Transaction and its properties
- Schedule and Recoverability
- Serializability and schedules
- Concurrency control
- Types of Locks
- Two phase locking
- Deadlock
- Time stamp based concurrency control
- Recovery Techniques
- Immediate update
- Deferred update
- Shadow paging

ACID Properties

Consistency:

Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database. This is typically the responsibility of the application programmer who codes the transactions.

Atomicity:

Either all operations of the transaction are reflected properly in the database, or none are.

Clearly lack of atomicity will lead to inconsistency in the database.

Isolation:

When multiple transactions execute concurrently, it should be the case that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently with it. The user view of a transaction system requires the isolation property, and the property that concurrent schedules take the system from one consistent state to another. These requirements are satisfied by ensuring that only serializable schedules of individually consistency preserving transactions are allowed.

Durability:

After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Need for concurrency control

Ensuring the isolation property of all concurrent transactions is the responsibility of a database management system. A way to execute concurrent transactions in serially. However, concurrent execution of transactions provides significant performance benefits.

Transaction and its properties:

Transaction: A transaction is unit of program execution that accesses and updates various data items. In general, a transaction is initiated by user program through high level data manipulation language or programming language (Example: SQL, C, Java etc.). where it delimits with start transaction and end transaction. Now the operations in between these two statements are executed as a transaction.

Transactions access data using two operations:

Read(X): Which transfers the data item X from the database to local buffer to execute the read operation.

Write(X): Which transfers the data item X from the local buffer of the transaction to write back to the database.

In real Database system, the write operation temporarily stored in memory and updates later on disk.

Example:

Bank transactions like credit, debit or transfer of amount from one account to another or updates on same account.

Let T_i be a transaction that transfers 5000 from account A to account B. Initially in account A 10000 and account B 20000 balance existed. This can be represented as:

Transaction ID	List of operations
T_i:	Start
	read(A);
	A:=A-5000;
	write(A);
	Read(B);
	B=B+5000;
	write(B);
	Stop;

Now the ACID properties should hold by transaction T_i :

Consistency: The database is consistent before and after transaction execution of T_i. the database remains consistent with sum of A and B at before and after transfer transaction executed. i.e

Initially before Transaction:

A=10000 and B=20000

A+B =10000+20000=30000

After Transaction (transfer of 5000 from A to B)

A=10000-5000=5000

Let Failure occurs at this point

Now A+B=5000+20000=25000.

Hence, the sum of database content befoe and after is not same as 30000 and 25000

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 k

Implicit integrity constraints

e.g. 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.

Atomicity: All operations in the transaction should be executed without any failure. Before execution of transaction T_i , the A nad B accounts with initial values as 10000 and 20000. Suppose during the transfer transaction a failure due to power failure, hardware and software errors will occurs. Suppose, after the write(A) and before write(B), a failure occurs then the values of A and B are 5000 and 20000. The system destroys 5000 as a result of this transaction. Therefore sum(A+B) after and before transactions are not consistent, then it leads to inconsistency.

Durability:

The durability property guarantees that, once the transaction completes successfully, all the updates on the database must be persistent, even if there is a failure after the transaction completes.

Ensuring durability is the responsibility of recovery management component. Hence the user has been notified about successful completion of transaction, it must be the case with

> Initially before Transaction:
>
> A=10000 and B=20000
>
> A+B =10000+20000=30000
>
> After Transaction (transfer of 5000 from A to B)
>
> A=10000-5000=5000
>
> **Let Failure occurs at this point**
>
> Now A+B=5000+20000=25000.
>
> Hence, the sum of database content before and after is not same as 30000 and 25000.

no system failure will result no loss of data corresponding to the transfer of funds.

Isolation:

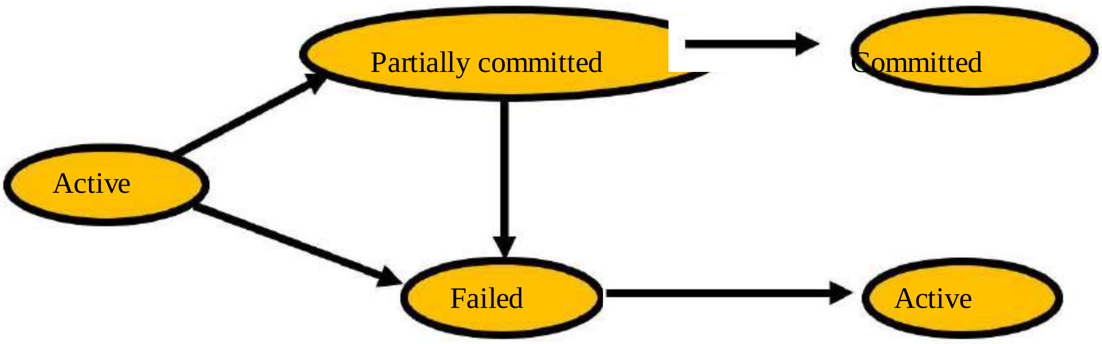
Isolation can be ensured trivially by running transactions serially that is, one after the other.

However, executing multiple transactions concurrently has significant benefits, as we will see later. For concurrent operations of multiple transactions leads to inconsistent state. Ensuring isolation is the responsibility of concurrency control component.

Let T_i and T_j are two transactions executed concurrently, their operations interleaved in desirable way resulting an inconsistent state.

Transaction State:

A transaction must be in one of the following states:



Active State:

The initial state of the transaction while it is executing.

Partially Committed:

After the final statement of the transaction has been executed.

Failed:

The transaction no longer proceed with normal execution, then it is in failed state.

Aborted:

After the transaction has been rolled back and the database has been restored to the prior to the state of the transaction. Two options after it has been aborted:

- Restart the transaction can be done only if no internal logical error
- Kill the transaction

Committed: After successful completion of the transaction.

4. Schedule and Recoverability

Schedule – 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 transaction must preserve the order in which the instructions appear in each individual transaction.

A transaction that successfully completes its execution will have a 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.

Concurrent executions:

Transaction processing system will allow multiple transactions to run concurrently. It leads to several problems like inconsistency of the data. Ensuring consistency of concurrent operations requires additional work to make serializable. Even though concurrent transactions has two major reasons:

Improved throughput and resource utilization.

Reduced waiting time.

Concurrency Control Schemes:

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)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

A serial schedule in which T_2 is followed by T_1 :

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

Schedule 1 and **schedule 2** are serial schedules. Each schedule consists various transactions, where series of instructions belonging to single transaction appear together in one schedule. **Schedule 3** is example of concurrent transaction. In this two transactions T_1 and T_2 running concurrently. In this the OS may execute a part from T_1 and switch to the second transactions T_2 and then switch back to the first transaction for some time and so on with multiple transactions. i.e. CPU time is shared among all the transactions

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)	
	read(B) $B := B + temp$ write(B)

Let T_1 and T_2 be the transactions defined previously. The following scheme is not a serial schedule, but it is *equivalent* to Schedule 1.

Schedule 4

T_1	T_2
read(A) A:=A-50	
	read(B) temp=A*0.1 A;=A-temp write(A) read(B)
write(A) read(B) B:=B+50 write(B)	
	B:=B+temp write(B)

In schedule 4, the CPU slicing is in different way to execute the transactions. It leads to the sum of A and B are different from before and after transactions as 950 and 2100. So this leads to inconsistent state.

5. Serializability and schedules

Basic Assumption – 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:**

A schedule is conflict serializable, if it is conflict equivalent to a serial schedule.

Let a schedule S, there are two consecutive operations I_i and I_j of transactions T_i and T_j . If I_i and I_j refers to different data items, then we can swap I_i and I_j.

If it refers the same data object then the order of two operations deal with four cases as given below.

I_i	I_j	
read(Q)	read(Q)	The order of I_i and I_j. does not matter
read(Q)	write(Q)	If I_i comes before I_j then it waits until I_j finish If I_j comes before I_i then no matter of order
write(Q)	read(Q)	Same as above
write(Q)	write(Q)	It does not matter order these two execution.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Shedule- 3

In the above schedule, the write(A) of T_1 conflicts with the read(A) of T_2 . Howerver write(A) of T_2 does not reflect with read(B) of T_1, because the two operations doest not refer the same data item.

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5 – schedule 3 after swapping of pair of instructions

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6 – A serial schedule euivallent to schedule 3

Conflicting Instructions

Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.

l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.

l_i = **read**(Q), l_j = **write**(Q). They conflict.

l_i = **write**(Q), l_j = **read**(Q). They conflict

l_i = **write**(Q), l_j = **write**(Q). They conflict

Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j 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

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

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.

Example of a schedule that is not conflict serializable:

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 >.

2. 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,

If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.

If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .

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.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

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.

What serial schedule is above equivalent to?

Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

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)

The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

Determining such equivalence requires analysis of operations other than read and write.

Recoverability:

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.

The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

T_8	T_9
read(A)	
write(A)	
	read(A)
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:

Cascading rollback – 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)

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 T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

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.

Implementation of Isolation:

Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.

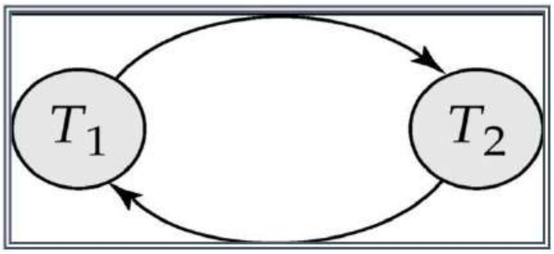
A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency.

Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.

Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

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)	
	$B := B + temp$
	write(B)

Testing for Serializability:



Consider some schedule of a set of transactions T_1, T_2, ..., T_n

Precedence graph — a direct graph where the vertices are the transactions (names).

We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.

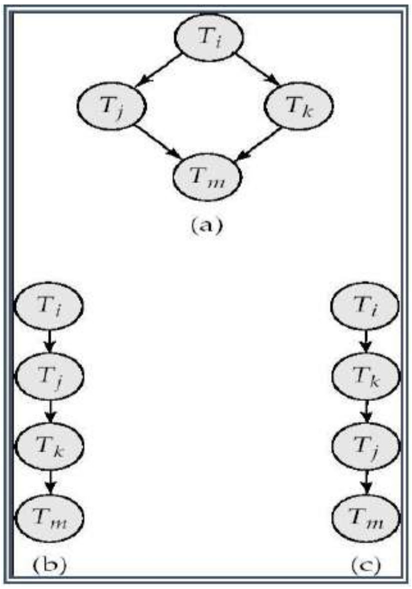
We may label the arc by the item that was accessed.

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 n^2 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 $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

Are there others?

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.

Concurrency Control:

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.

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

- E.g. a read-only transaction that wants to get an approximate total balance of all Accounts.

  E.g. database statistics computed for query optimization can be approximate (why?)
- 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.

Transaction Definition in SQL Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.

In SQL, a transaction begins implicitly.

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

E.g. in JDBC, connection.setAutoCommit(false);

Types of Locks

There are various modes to lock data items. They are

Shared(S): If a transaction Ti has shared mode lock on data item Q then Ti can read but not write Q. **lock-S(Q)** instruction is used in shared mode.

Exclusive(X): If a transaction has obtained an exclusive mode lock on data item Q, then Ti can perform both read and write. **lock-X(Q)** instruction is used to lock in exclusive mode.

A lock is a mechanism to control concurrent access to a data item. 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:

***T1*:**	***T2*:**	***T3*:**	***T4*:**
lock-X*(B)*; **read *(B)*;** **B:=B-50;** **write(B);** **unlock*(B)*;** **lock-X*(A)*;** **read *(A)*;** **A:=A+50;** **write(A);** **unlock*(A)*;**	**lock-S*(A)*;** **read *(A)*;** **unlock*(A)*;** **lock-S*(B)*;** **read *(B)*;** **unlock*(B)*;** **display*(A+B)***	**lock-X*(B)*;** **read *(B)*;** **B:=B-50;** **write(B);** **lock-X*(A)*;** **read *(A)*;** **A:=A+50;** **write(A);** **unlock*(B)*;** **unlock*(A)*;**	**lock-S*(A)*;** **read *(A)*;** **lock-S*(B)*;** **read *(B)*;** **display*(A+B);*** **unlock*(A)*;** **unlock*(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.

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. Consider the partial schedule

Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on *A*. Such a situation is called a **deadlock**. To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released. The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

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.

Two phase locking

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 (e.g., ordering of access to data), two-hase locking is needed for conflict serializability in the following sense:

Given a transaction Ti 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)

Two-Phase Locking Protocol

This protocol ensures conflict-serializable schedules.

Phase 1: Growing Phase

transaction may obtain locks

transaction may not release locks

Phase 2: Shrinking Phase

transaction may release locks

transaction may not obtain locks

The 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/

This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

Automatic Acquisition of Locks :

A transaction Ti 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

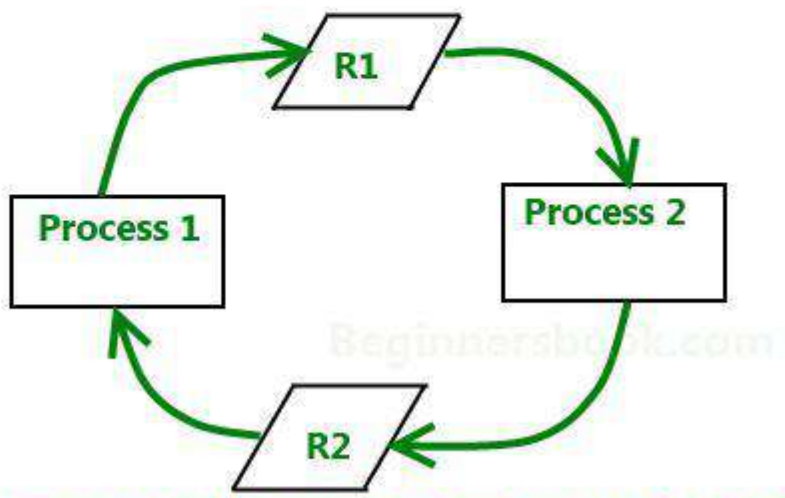
A lock manager can be implemented as a separate process to which transactions send lock and unlock requests

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

The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Deadlock: A **deadlock** is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.



Process P1 holds resource R2 and requires R1 while Process P2 holds resource R1 and requires R2.

10. Time stamp based concurrency control

Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j). 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 T_i issues a **read**(Q)

- If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.

  If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to **max**(R-timestamp(Q), TS(T_i)).Suppose that transaction T_i issues **write**(Q).

  If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced.

  If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

Recovery Techniques

To see where the problem has occurred we generalize the failure into various categories, as follows:

Transaction failure: When a transaction is failed to execute or it reaches a point after which it cannot be completed successfully it has to abort. This is called transaction failure. Where only few transaction or process are hurt.

Recovery and Atomicity:

Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.

Consider transaction T_i that transfers \$50 from account A to account B; goal is either to perform all database modifications made by T_i or none at all.

Several output operations may be required for T_i (to output A and B). A failure may occur after one of these modifications have 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. Two approaches for recovery are **log-based recovery**, and **shadow-paging**. Assume (initially) that transactions run serially, that is, one after the other.

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.

Log-Based Recovery:

A **log** is kept on stable storage.

The log is a sequence of **log records**, and maintains a record of update activities on the database. Log record has 3 fields:

Transaction Identifier: Unique identifier of the transaction that performed write operation.

Data item identifier: Unique identification of the data item written

Old value: Value of the item prior to the write

New value: Value of the item after write transaction

Various log records are:

<T_i **start**> log record *Before* T_i executes **write**(X),

<T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X. Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.

<T_i **commi**t> Transaction T_i has committed

<T_i **abort**> Transaction T_i has aborted

Deferred database modification

Immediate database modification

12. Immediate update

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. 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.

Recovery procedure has two operations instead of one:

undo(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i

redo(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

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 T_i needs to be undone if the log contains the record

<T_i **start**>, but does not contain the record <T_i **commit**>.

Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

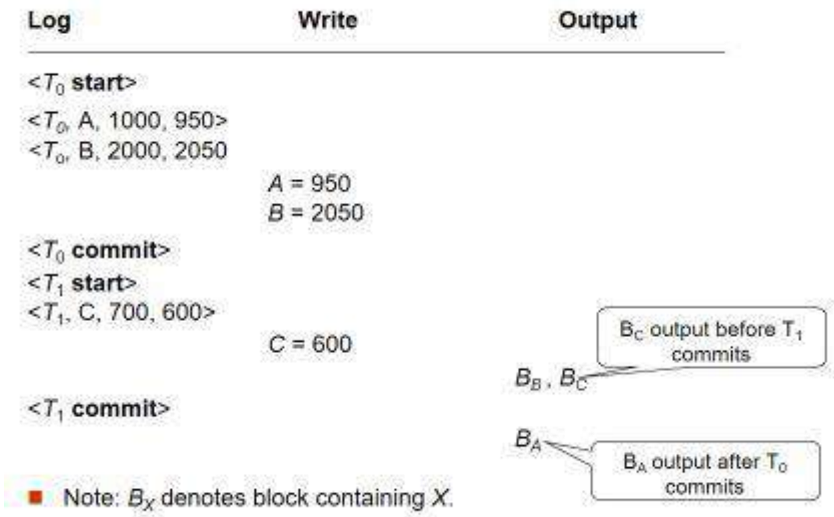
Undo operations are performed first, then redo operations.

Example: **Immediate Database Modification**

Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken example transactions T_0 and T_1 (T_0 executes before T_1):

T_0:	T_1 :
read (*A*)	**read** (*C*)
- A - 50	*C:-C- 100*
Write (*A*)	**write** (*C*)
read (*B*)	
B:- B + 50	
write (*B*)	

Let accounts A, B and C initially has 1000, 2000 and 700 respectively. The log entry of both the transactions are:



Below we show the log as it appears at three instances of time. Recovery actions in each case above are:

undo (T_0): B is restored to 2000 and A to 1000.

undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

13. Deferred update

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

<T_i ***start***>*transaction* T_i started.

A **write**(X) operation results in a log record :

<T_i, 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 T_i partially commits,

<T_i **commit**> is written to the log

Finally, the log records are read and used to actually execute the previously deferred writes. During recovery after a crash, a transaction needs to be redone if and only if both

<T_i **start**> and<T_i **commit**> are there in the log.

Redoing a transaction T_i

< redoT_i> sets the value of all data items updated by the transaction to the new values.

Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken example transactions T_0 and T_1 (T_0 executes before T_1):

T_0:	T_1 :
read (*A*) *- A - 50* **Write** (*A*) **read** (*B*) *B:- B + 50* **write** (*B*)	**read** (*C*) *C:-C- 100* **write** (*C*)

Let accounts A,B and C initially has 1000, 2000 and 700 respectively. The log entry of both the transactions are:

<T_0 ***start***>

<T_0***, A, 950***>

<T_0***, B, 2050***>

<T_0***, commit***>

<T_1 ***start***>

<T_1***, C, 600***>

<T_1***, commit***>

14. Shadow paging

Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially

Idea: maintain two page tables during the lifetime of a transaction –the current page table, and the shadow page table

Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.

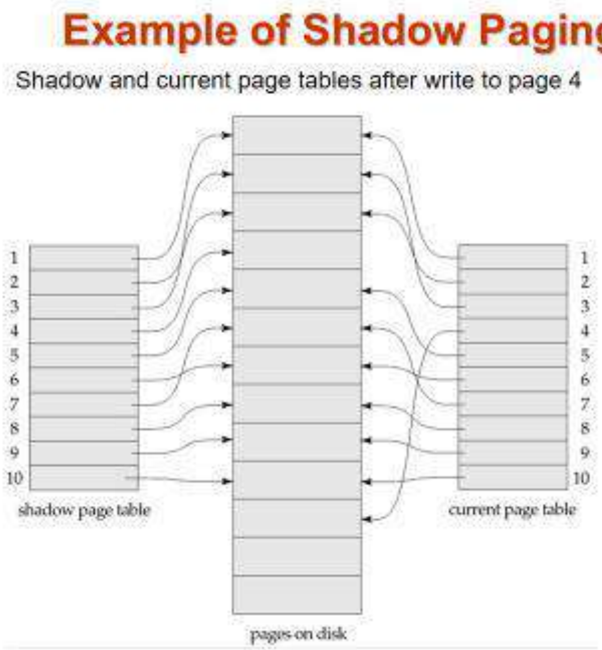
Shadow page table is never modified during execution

To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.

Whenever any page is about to be written for the first time, A copy of this page is made onto an unused page.

The current page table is then made to point to the copy

The update is performed on the copy



To commit a transaction :

Flush all modified pages in main memory to disk

Output current page table to disk

Make the current page table the new shadow page table, as follows:

keep a pointer to the shadow page table at a fixed (known) location on disk.

to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk

Once pointer to shadow page table has been written, transaction is committed.

No recovery is needed after a crash — new transactions can start right away, using the shadow page table.

Pages not pointed to from current/shadow page table should be freed (garbage collected).

Advantages of shadow-paging over log-based schemes

no overhead of writing log records

recovery is trivial

Disadvantages :

Copying the entire page table is very expensive

Can be reduced by using a page table structured like a B^+-tree

No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes

Commit overhead is high even with above extension

Need to flush every updated page, and page table

Data gets fragmented (related pages get separated on disk)

After every transaction completion, the database pages containing old versions of modified data need to be garbage collected

Hard to extend algorithm to allow transactions to run concurrently

Easier to extend log based schemes

UNIT-V

Data Storage and Query Processing

Record storage and primary file organization

Secondary storage devices

Operations on files

Heap File

Sorted files

Hashing techniques

Index structures for files

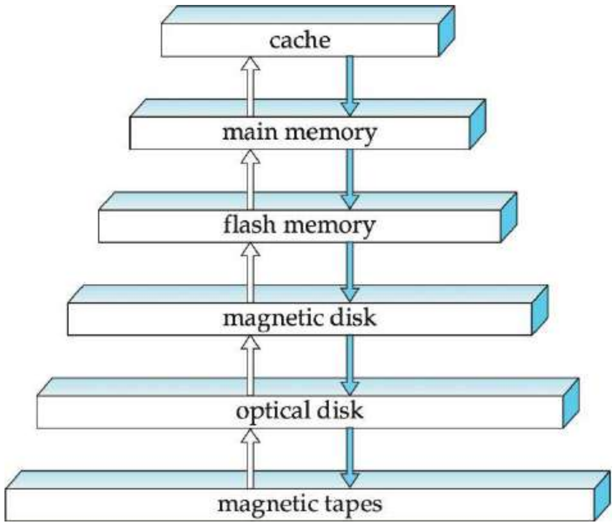
Different types of indexes

B tree and B+ tree

Query processing

Record storage and primary file organization

Storage Hierarchy



primary storage: Fastest media but volatile (cache, main memory).

secondary storage: next level in hierarchy, non-volatile, moderately fast access time

also called on-line storage

E.g. flash memory, magnetic disks

tertiary storage: lowest level in hierarchy, non-volatile, slow access time

also called off-line storage

E.g. magnetic tape, optical storage

File organization: Method of arranging a file of records on external storage. Record id (rid) is sufficient to physically locate record.

The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields.

One approach:

- assume record size is fixed
- each file has records of one particular type only
- different files are used for different relations

This case is easiest to implement; will consider variable length records later.

Fixed-Length Records

Store record i starting from byte n * (i – 1), where n is the size of each record.Record access is simple but records may cross blocks.

Modification: do not allow records to cross block boundaries

Deletion of record i: alternatives:

move records i + 1, . . ., n to i, . . . , n – 1

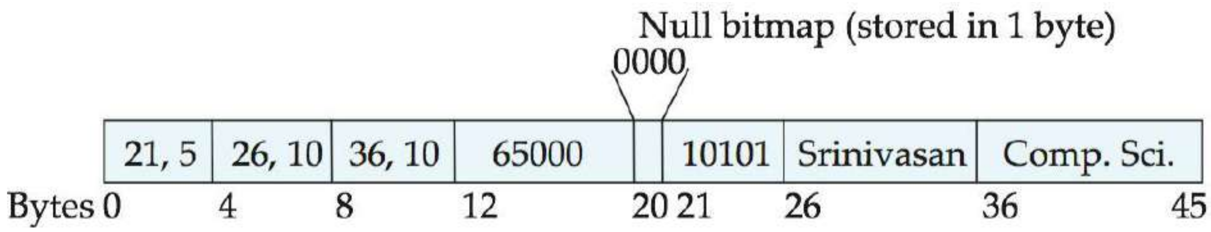
move record n to i

do not move records, but link all free records on a free list

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records

Variable-length records arise in database systems in several ways:
Storage of multiple record types in a file.
Record types that allow variable lengths for one or more fields such as strings (varchar)
Record types that allow repeating fields (used in some older data models).
Attributes are stored in order
Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
Null values represented by null-value bitmap



Organization of Records in Files

Heap – a record can be placed anywhere in the file where there is space

Sequential – store records in sequential order, based on the value of the search key of each record

Hashing – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

Records of each relation may be stored in a separate file. In a multitable clustering file organization records of several different relations can be stored in the same file

Motivation: store related records on the same block to minimize I/O

Operations on files

Secondary storage devices

Disks: Can retrieve random page at fixed cost.But reading several consecutive pages is much cheaper than reading them in random order

Tapes: Can only read pages in sequence. Cheaper than disks; used for archival storage.

Sequential File Organization

Suitable for applications that require sequential processing of the entire file

The records in the file are ordered by a search-key

Deletion – use pointer chains

Insertion –locate the position where the record is to be inserted

if there is free space insert there

if no free space, insert the record in an overflow block

In either case, pointer chain must be updated

Need to reorganize the file from time to time to restore sequential order

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

Indexes are data structures that allow us to find the record ids of records with given values in index search key fields

Architecture: Buffer manager stages pages from external storage to main memory buffer pool. File and index layers make calls to the buffer manager.

Alternative File Organizations:

Many alternatives exist, *each ideal for some situations, and not so good in others:*

Heap (random order) files: Suitable when typical access is a file scan retrieving all records.

Sorted Files: Best if records must be retrieved in some order, or only a `range' of records is needed.

Indexes: Data structures to organize records via trees or hashing.

Like sorted files, they speed up searches for a subset of records, based on values in certain ("search key") fields. Updates are much faster than in sorted files.

Primary and secondary Indexes:

Primary vs. secondary: If search key contains primary key, then called primary index.

Unique index: Search key contains a candidate key.

Clustered and uncluttered:

If order of data records is the same as, or `close to', order of data entries, then called clustered index.

Alternative 1 implies clustered; in practice, clustered also implies Alternative 1 (since sorted files are rare).

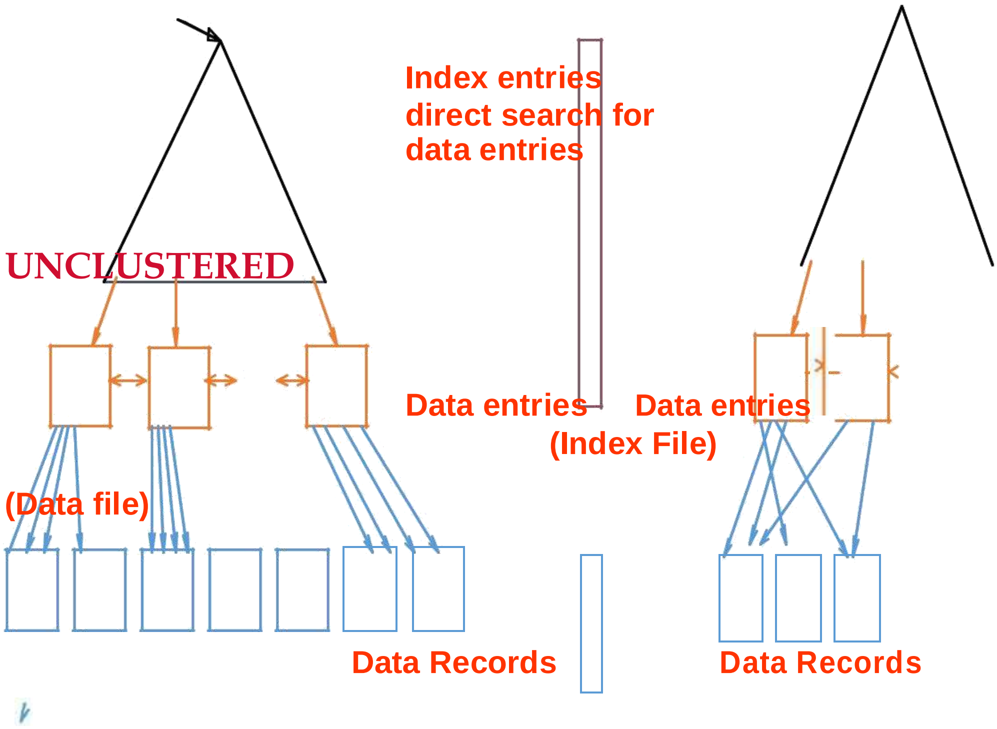
A file can be clustered on at most one search key.

Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Index

Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.

To build clustered index, first sort the Heap file (with some free space on each page for future inserts).



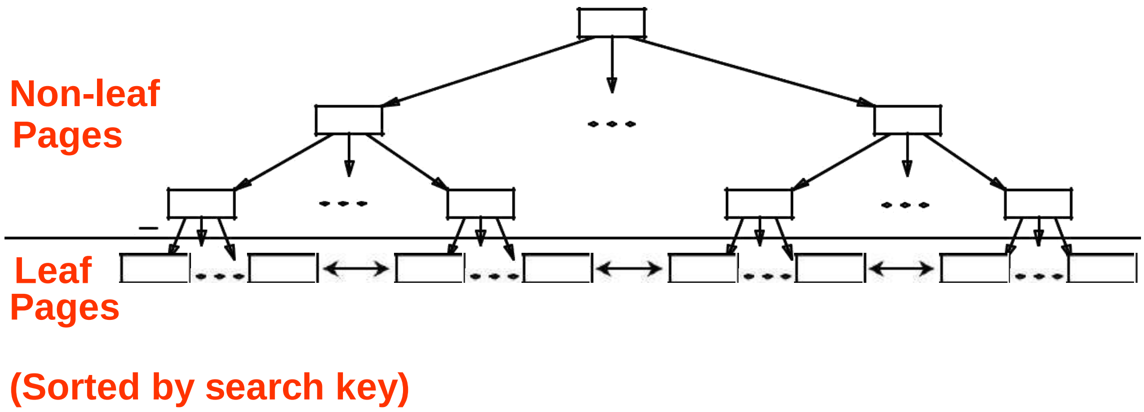
Overflow pages may be needed for inserts. to', but (Thus, order of data recs is `close not identical to, the sort order.)

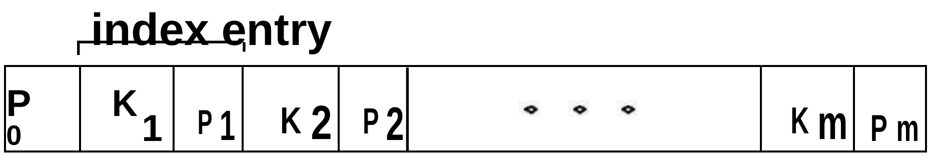
Index Data Structures:

An *index* on a file speeds up selections on the *search key fields* for the index.

Any subset of the fields of a relation can be the search key for an index on the relation.

Search key is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).





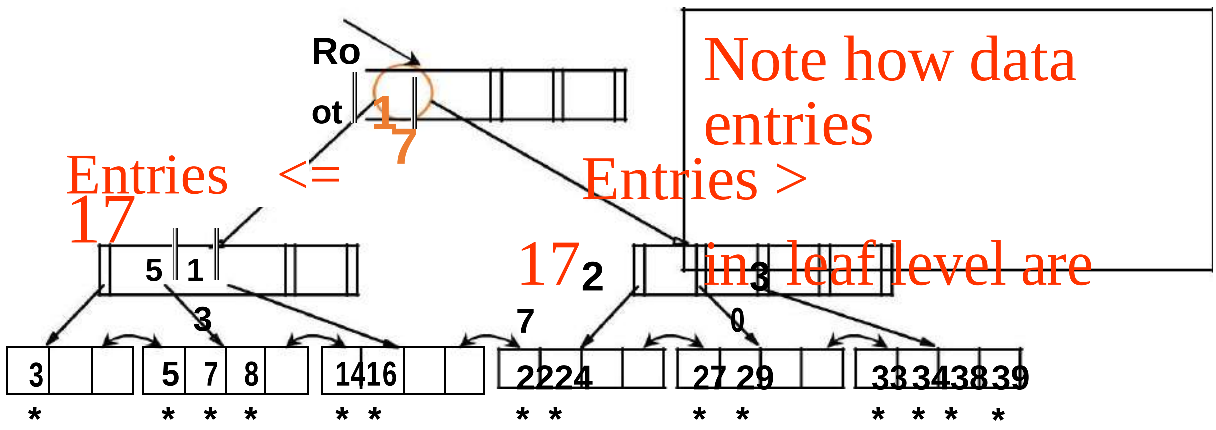
An index contains a collection of *data entries*, and supports efficient retrieval of all data entries **k*** with a given key value **k**.

Given data entry k*, we can find record with key k in at most one disk I/O.

(Details soon …)

B+ Tree Indexes

Example B+ Tree



Find 28*? 29*? All > 15* and < 30*

Insert/delete: Find data entry in leaf, then change it. Need to adjust parent sometimes.

–And change sometimes bubbles up the tree

Hash-Based Indexing:

Hash-Based Indexes

Good for equality selections.

Index is a collection of *buckets*.

- Bucket = *primary* page plus zero or more *overflow* pages.
- Buckets contain data entries.

Hashing function **h**: **h**(*r*) = bucket in which (data entry for) record *r* belongs. **h** looks at the *search key* fields of *r*. *No need for "index entries" in this scheme.*

Alternatives for Data Entry **k*** in Index

In a data entry k* we can store:

- Data record with key value **k,** or
- <**k**, rid of data record with search key value **k**>, or
- <**k**, list of rids of data records with search key **k**>

Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.

Tree Based Indexing:

–Examples of indexing techniques: B+ trees, hash-based structures

–Typically, index contains auxiliary information that directs searches to the desired data entries

Alternative 1:

–If this is used, index structure is a file organization for data records (instead of a Heap file or sorted file).

–At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records are duplicated, leading to redundant storage and potential inconsistency.)

–If data records are very large, # of pages containing data entries is high.

Implies size of auxiliary information in the index is also large, typically.

Alternatives 2 and 3:

–Data entries typically much smaller than data records. So, better than

Alternative 1 with large data records, especially if search keys are small. (Portion of index structure used to direct search, which depends on size of data entries, is much smaller than with Alternative 1.)

–Alternative 3 more compact than Alternative 2, but leads to variable sized data entries even if search keys are of fixed length.

Cost Model for Our Analysis

We ignore CPU costs, for simplicity:

– **B:** The number of data pages

– **R:** Number of records per page

– **D:** (Average) time to read or write disk page

– Measuring number of page I/O's ignores gains of pre-fetching a sequence of pages; thus, even I/O cost is only approximated.

– Average-case analysis; based on several simplistic assumptions.

Comparison of File Organizations:

Heap files (random order; insert at eof)

Sorted files, sorted on <*age, sal*>

Clustered B+ tree file, Alternative (1), search key <*age, sal*>

Heap file with unclustered B + tree index on search key <*age, sal*>

Heap file with unclustered hash index on search key <*age, sal*>

Operations to Compare

Scan: Fetch all records from disk

Equality search

Range selection

Insert a record

Delete a record

Assumptions in Our Analysis

Heap Files:

– Equality selection on key; exactly one match.

Sorted Files:

– Files compacted after deletions.

Indexes:

– Alt (2), (3): data entry size = 10% size of record

– Hash: No overflow buckets.

• 80% page occupancy => File size = 1.25 data size

– Tree: 67% occupancy (this is typical).

• Implies file size = 1.5 data size

Scans:

– Leaf levels of a tree-index are chained.

– Index data-entries plus actual file scanned for unclustered indexes.

Range searches:

We use tree indexes to restrict the set of data records fetched, but ignore hash indexes.

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog $_2$B	D(log $_2$ B + # pgs with match recs)	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog $_F$ 1.5B	D(log $_F$ 1.5B + # pgs w. match recs)	Search + D	Search +D
(4) Unclust. Tree index	BD(R+0.15)	D(1 + log $_F$ 0.15B)	D(log $_F$ 0.15B + # pgs w. match recs)	Search + 2D	Search + 2D
(5) Unclust. Hash index	BD(R+0.125)	2D	BD	Search + 2D	Search + 2D

Understanding the Workload

For each query in the workload:

Which relations does it access?

Which attributes are retrieved?

Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

For each update in the workload:

Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?

The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Choice of Indexes

What indexes should we create?

Which relations should have indexes? What field(s) should be the search key?

Should we build several indexes?

For each index, what kind of an index should it be?

Clustered? Hash/tree?

One approach: Consider the most important queries in turn. Consider the best plan using the current indexes, and see if a better plan is possible with an additional index. If so, create it.

– Obviously, this implies that we must understand how a DBMS evaluates queries and creates query evaluation plans!

– For now, we discuss simple 1-table queries.

Before creating an index, must also consider the impact on updates in the workload!

– Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.

Index Selection Guidelines

Attributes in WHERE clause are candidates for index keys.

Exact match condition suggests hash index.

Range query suggests tree index.

Clustering is especially useful for range queries; can also help on equality queries if there are many duplicates.

Multi-attribute search keys should be considered when a WHERE clause contains several conditions.

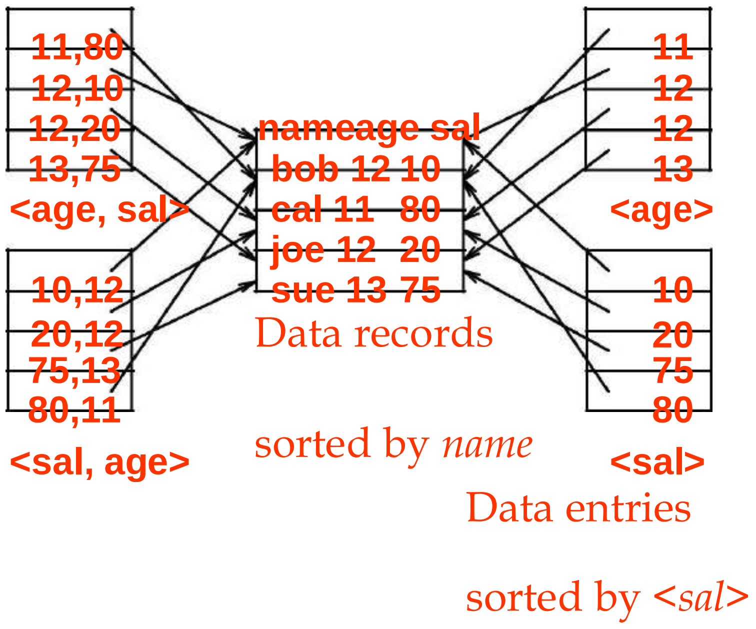
Order of attributes is important for range queries.

Such indexes can sometimes enable index-only strategies for important queries.

For index-only strategies, clustering is not important!

Examples of composite key

indexes using lexicographic order.



Composite Search Keys: Search on a combination of fields.

Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:

age=20 and sal =75

Range query: Some field value is not a constant. E.g.:

age =20; or age=20 and sal > 10

Data entries in index sorted by search key to support range queries.

Lexicographic order, or Spatial order.

Composite Search Keys

To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age*,*sal*> would be better than an index on *age* or an index on *sal*.

Choice of index key orthogonal to clustering etc.

If condition is: 20<*age*<30 AND 3000<*sal*<5000:

Clustered tree index on <*age*,*sal*> or <*sal*,*age*> is best.

If condition is: *age*=30 AND 3000<*sal*<5000:

Clustered <*age*,*sal*> index much better than <*sal*,*age*> index!

Composite indexes are larger, updated more often.

Index-Only Plans

A number of queries can be answered without retrieving any tuples from one or more of the relations involved if a suitable index is available.

Summary

Many alternative file organizations exist, each appropriate in some situation.

If selection queries are frequent, sorting the file or building an *index* is important.

Hash-based indexes only good for equality search.

Sorted files and tree-based indexes best for range search; also good for equality search. (Files rarely kept sorted in practice; B+ tree index is better.)

Index is a collection of data entries plus a way to quickly find entries with given key values.

Data entries can be actual data records, <key, rid> pairs, or <key, rid-list> pairs.

– Choice orthogonal to *indexing technique* used to locate data entries with a given key value.

Can have several indexes on a given file of data records, each with a different search key.

Indexes can be classified as clustered vs. unclustered, primary vs. secondary, and dense vs. sparse. Differences have important consequences for utility/performance.

As for any index, 3 alternatives for data entries **k***:

Data record with key value **k**

<**k**, rid of data record with search key value **k**>

<**k**, list of rids of data records with search key **k**>

Choice is orthogonal to the *indexing technique* used to locate data entries **k***.

Different types of indexes

Indexing mechanisms used to speed up access to desired data.

E.g., 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:

Ordered indices: search keys are stored in sorted order

Hash indices: search keys are distributed uniformly across "buckets" using a "hash function". **Index Evaluation Metrics**

Access types supported efficiently. E.g., records with a specified value in the attribute or records with an attribute value falling in a specified range of values.

Access time

Insertion time

Deletion time

Space overhead

Ordered indices: In an ordered index, index entries are stored sorted on the search key value. E.g., author catalog in library.

Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file. Also called clustering index. The 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.

Hash Function:

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:

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

E.g. h(Music) = 1 h(History) = 2

h(Physics) = 3 h(Elec. Eng.) = 3

Hash file organization of *instructor* file, using *dept_name* as key bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash Indices:

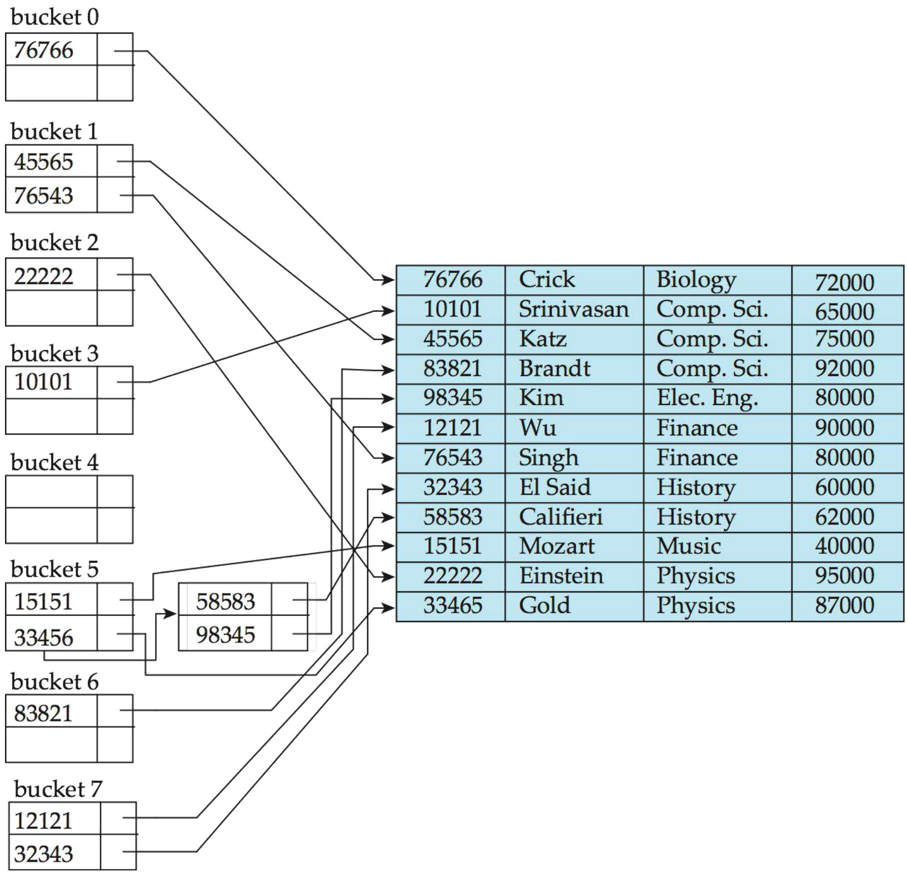
Hashing can be used not only for file organization, but also for index-structure creation.

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.



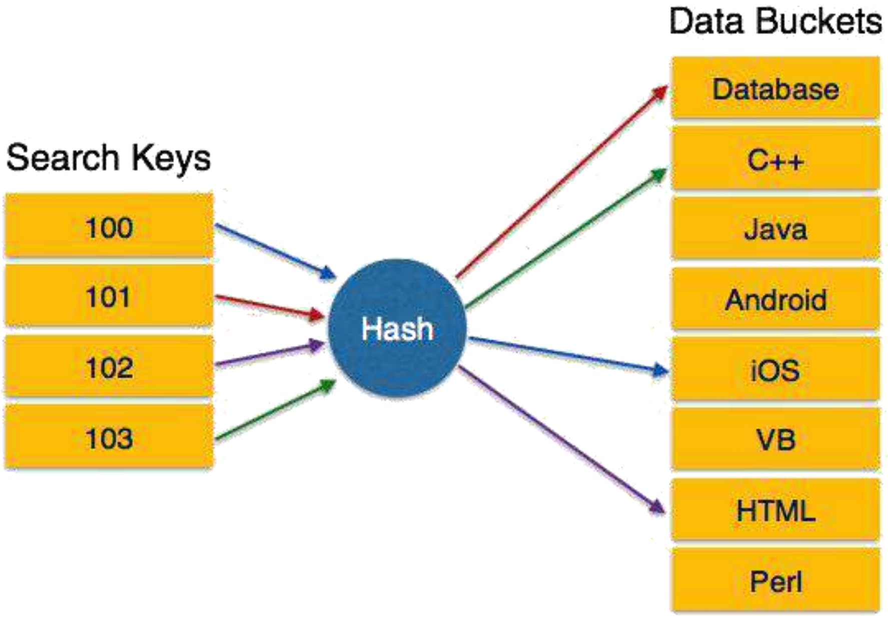
Hash Based Indexing:

Bucket: Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.

Hash Function: A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keyto bucket addresses.

Static Hashing:

In static hashing, when a search-key value is provided the hash function always computes the same address. For example, if mod-4 hash function is used then it shall generate only 5 values. The output address shall always be same for that function. The numbers of buckets provided remain same at all times.



[*Image: Static Hashing*]

Operation:

Insertion: When a record is required to be entered using static hash, the hash function h, computes the bucket address for search key K, where the record will be stored.

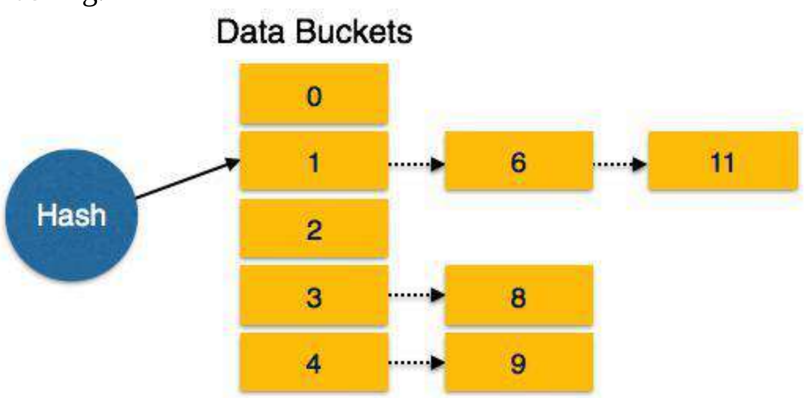
Bucket address = h(K)

Search: When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
Delete: This is simply search followed by deletion operation.

Bucket Overflow:

The condition of bucket-overflow is known as collision. This is a fatal state for any static hash function. In this case overflow chaining can be used.

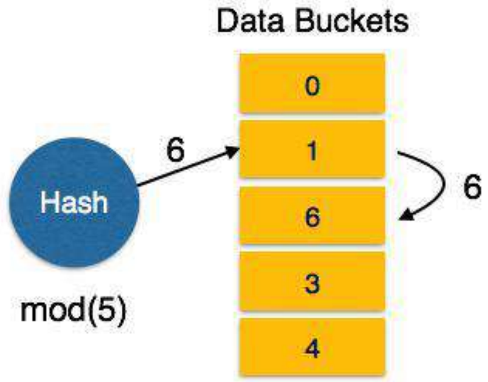
Overflow Chaining: When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.



[*Image: Overflow chaining*]

Linear Hashing:

Linear Probing: When hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.



[*Image: Linear Probing*]

For a hash function to work efficiently and effectively the following must match:

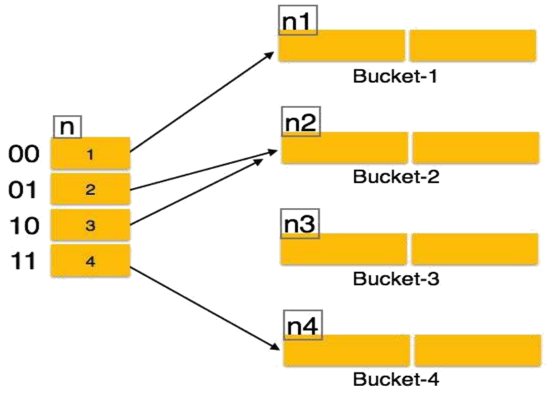
Distribution of records should be uniform
Distribution should be random instead of any ordering

Extendable Hashing:

Dynamic Hashing

Problem with static hashing is that it does not expand or shrink dynamically as the size of database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing.

Hash function, in dynamic hashing, is made to produce large number of values and only a few are used initially.



[*Image: Dynamic Hashing*]

Organization

The prefix of entire hash value is taken as hash index. Only a portion of hash value is used for computing bucket addresses. Every hash index has a depth value, which tells it how many bits are used for computing hash function. These bits are capable to address 2n buckets. When all these bits are consumed, that is, all buckets are full, then the depth value is increased linearly and twice the buckets are allocated.

Operation

Querying: Look at the depth value of hash index and use those bits to compute the bucket address.

Update: Perform a query as above and update data.

Deletion: Perform a query to locate desired data and delete data.

Insertion: compute the address of bucket

If the bucket is already full

Add more buckets

Add additional bit to hash value

Re-compute the hash function

Else

Add data to the bucket

If all buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and queries require range of data. When data is discrete and random, hash performs the best.

Hashing algorithm and implementation have high complexity than indexing. All hash operations are done in constant time.

Extendable Vs. Linear Hashing:

Benefits of extendable hashing:

hash performance doesn't degrade with growth of file

minimal space overhead

Disadvantages of extendable hashing:

extra level of indirection (bucket address table) to find desired record

bucket address table may itself become very big (larger than memory)

- need a tree structure to locate desired record in the structure!

Changing size of bucket address table is an expensive operation

Linear hashing: is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

B tree and B+ tree

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.

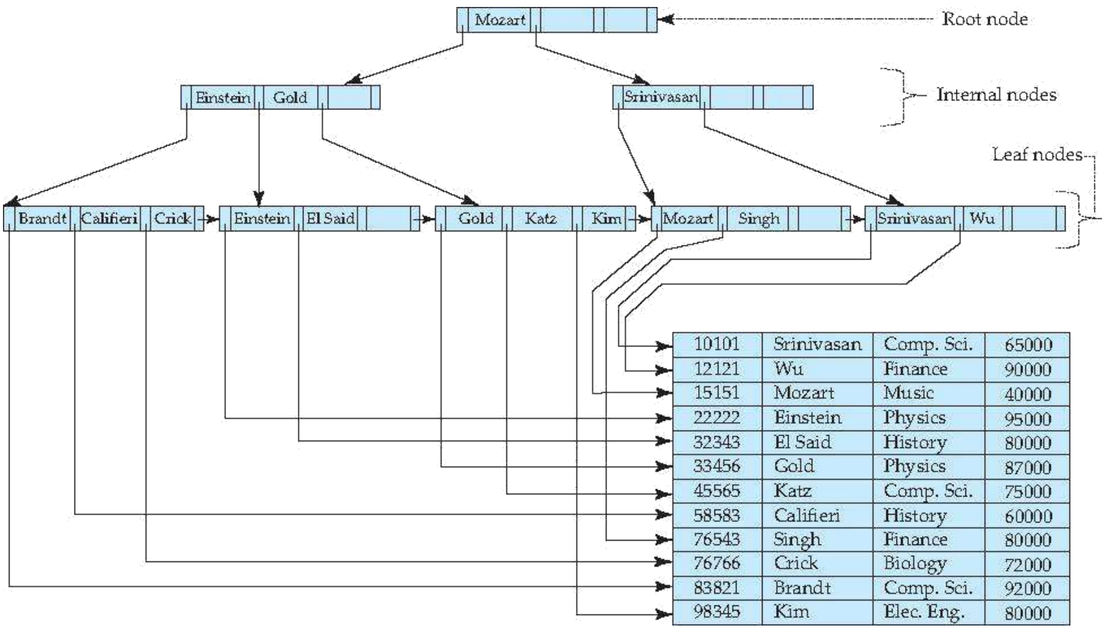
(Minor) disadvantage of B^+-trees:

Extra insertion and deletion overhead, space overhead.

Advantages of B^+-trees outweigh disadvantages

B^+-trees are used extensively

Example of B+Tree:



B+-tree properties:

All paths from root to leaf are of the same length

Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.

A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values

Special cases: If the root is not a leaf, it has at least 2 children.

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

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

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

$K_1 < K_2 < K_3 < \ldots < K_{n-1}$

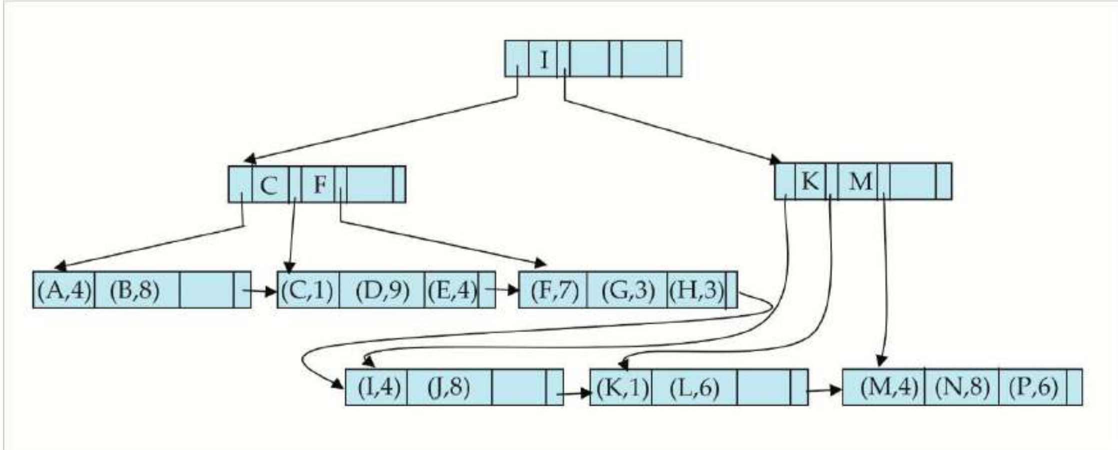
(Initially assume no duplicate keys, address duplicates later)

Properties of a leaf node:

For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,

If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values

P_n points to next leaf node in search-key order



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 entries

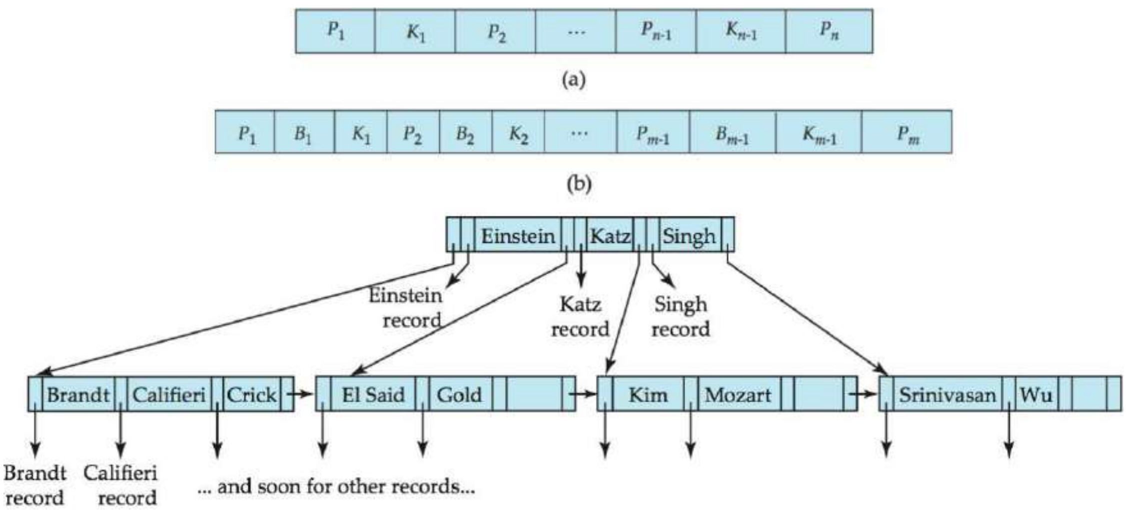
B-Tree Index Files

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

Non leaf node – pointers Bi are the bucket or file record pointers



B – tree indexing

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 out weigh disadvantages.

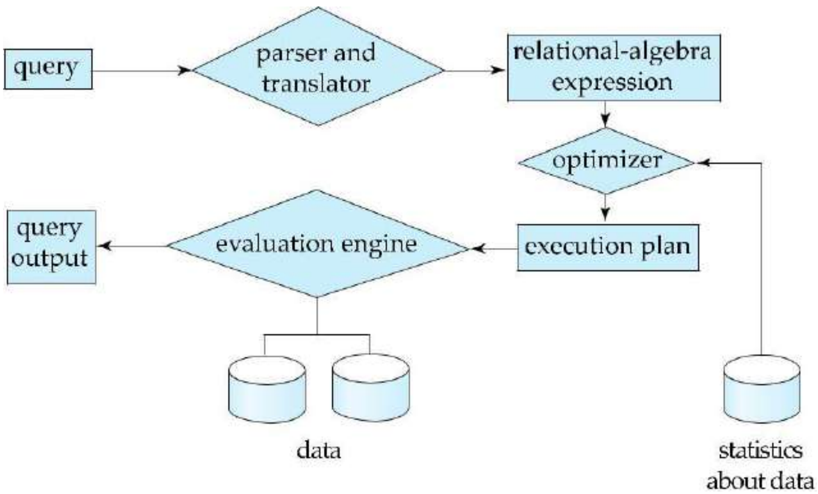
Query processing

Basic steps in Query Processing:

Parsing and translation

Optimization

Evaluation



Parsing and translation

Translate the query into its internal form. This is then translated into relational algebra.

Parser checks syntax, verifies relations

Evaluation

The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

Optimization

A relational algebra expression may have many equivalent expressions

E.g.,

$\sigma_{salary<75000}(\prod_{salary}(instructor))$ is equivalent to

$\prod_{salary}(\sigma_{salary<75000}(instructor))$

Each relational algebra operation can be evaluated using one of several different algorithms

Correspondingly, a relational-algebra expression can be evaluated in many ways.

Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

E.g., can use an index on *salary* to find instructors with salary < 75000,

or can perform complete relation scan and discard instructors with salary ≥ 75000

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost. Cost is estimated using statistical information from the database catalog. e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

Cost is generally measured as total elapsed time for answering query

a. Many factors contribute to time cost

disk accesses, CPU, or even network *communication*

Typically disk access is the predominant cost, and is also relatively easy to estimate.

a. Number of seeks * average-seek-cost
b. Number of blocks read * average-block-read-cost

Number of blocks written * average-block-write-cost

Cost to write a block is greater than cost to read a block

data is read back after being written to ensure that the write was successful

For simplicity we just use the **number of block transfers** *from disk and the* **number of seeks** as the cost measures

t_T – time to transfer one block

t_S – time for one seek

Cost for b block transfers plus S seeks

$* t_T + S * t_S$

- We ignore CPU costs for simplicity

Real systems do take CPU cost into account

We do not include cost to writing output to disk in our cost formulae