SDSC 5003
Introduction

Instructor: Yu Yang
yuyang@cityu.edu.hk

What is Data Storing and Retrieving?

- Data Management (Both **Engineering** and Science)
- Database
- How to Design and Use Databases

What is a Database?

- A **database** contains information that is relevant to some enterprise
  - The main goal of a database is to store and retrieve this information, e.g. CityU (student, faculty, course, classroom, office…)
- Databases typically contain **large amounts** of information
- It should be possible to access this information efficiently and securely, e.g. Amazon has 40+ categories of products, 10K+ sub-categories, > 12 million products, over 100 million users …

Course Website

- https://canvas.cityu.edu.hk/courses/61395
- Instructor: Yu Yang, yuyang@cityu.edu.hk, LOW Tsz Kin dave.lo@cityu.edu.hk
- Office hours: 2:00pm-3:00pm Monday, LAU-16-283, starting from week 2

Outline

- What is a Database?
- Why we need Databases?
- Database overview

What is a Database System?

- A database system consists of two components
  - Database (DB) and
  - Database Management System (DBMS)
- The DB contains the data
- The DBMS is software that stores, manages and retrieves the information in the DB

A Survey

- What is your undergraduate major?
  - Computer Science (including Software Engineering, IoT and other closed related majors)
  - Engineering (Industrial/System/Electrial & Electronic/Mechanical/…)
  - Math/Statistics/Physics
  - Business
  - Others

What is a Database?

Why We Need Databases?

Data in the Current Millenium

- Consider Walmart
  - Which handles more than 1 million customer transactions per hour
  - Imported into databases estimated to contain more than 2.5 petabytes (2,560 terabytes) of data
    - Gigabyte – 2^{30} bytes
    - Terabyte – 2^{40} bytes
    - Petabyte – 2^{50} bytes

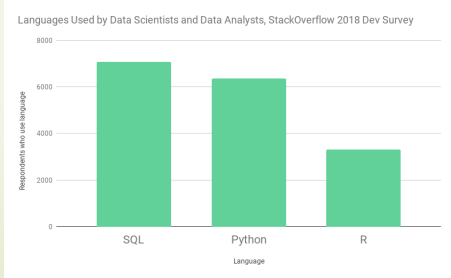
Customer Transactions

Database Applications

- Any application that has to store large amounts of data probably needs a database
  - Banking
  - Airlines
  - Universities
  - Credit card transactions
  - Finances
  - Sales
  - On-line retailers
  - Manufacturing
  - Human resources
  - MMORPGs
  - …

Database to Data Scientists

- Massive data
- Preparing Data often takes 80% of time



Data Storage Without DBMS

- Different data sources and data formats
  - Parsing different data formats is tedious!

CSV

ID	Gender	City	Monthly_I
ID000002C	Female	Delhi	20000
ID000004E	Male	Mumbai	35000
ID000007H	Male	Panchkula	22500
ID000008I	Male	Saharsa	35000
ID000009J	Male	Bengaluru	100000
ID000010K	Male	Bengaluru	45000
ID000011L	Female	Sindhudui	70000
ID000012M	Male	Bengaluru	20000
ID000013N	Male	Kochi	75000
ID000014O	Female	Mumbai	30000
ID000016Q	Male	Mumbai	25000
ID000018S	Female	Surat	25000
ID000019T	Female	Pune	24000
ID000021V	Male	Bhubanes	27000
ID000022W	Female	Howrah	28000

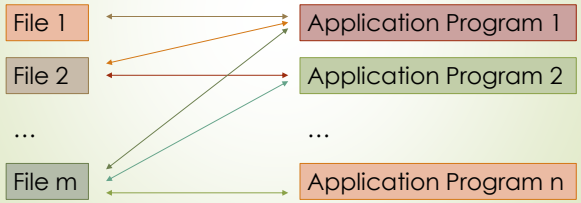
JSON

```
{
  "Employee": [
    {
      "id":"1",
      "Name": "Ankit",
      "Sal": "1000",
    },
    {
      "id":"2",
      "Name": "Faizv",
```

```
<?xml version="1.0"?>
<contact-info>
<name>Ankit</name>
<company>Anlytics Vidhya</company>
<phone>+9187654321</phone>
</contact-info>
```

Data Storage Without DBMS

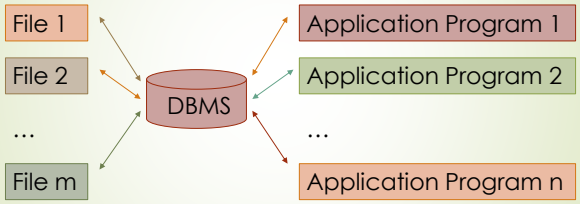
- Data would be collected in many different files and
- Used by many application programs



What Happens If …

- An attribute is added to one of the files?
- Information that is in more than one file is changed by a program that only interacts with one file?
- We need to access a single record out of millions of records?
- Several programs need to access *and modify* the same record at the same time?
- The system crashes while one of the application programs is running?

Data Storage with a DBMS



DBMS Advantages

- All access to data is centralized and managed by the DBMS which provides
  - Logical data independence
  - Physical data independence
  - Reduced application development time
  - Efficient access
  - Data integrity and security
  - Concurrent access and concurrency control
  - Crash recovery

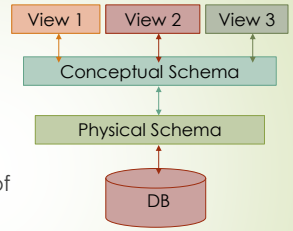
Database Overview

Data Models

- A *database* models a real-world enterprise, e.g. CityU
- A *data model* is a formal language for describing data
- A *schema* is a description of a particular collection of data using a particular data model
- The most widely used data model is the *relational data model*
  - The main concept ***relation***, essentially a table with rows and columns
  - Each relation has a schema, e.g. Enrolled(sid: string, cid: string, gpa: real)

Data Abstraction

- Physical schema
  - Describes how data are stored and indexed
- Conceptual (or logical) schema
  - Describes data in terms of the data model
- External (or **view**) schema
  - Describes how some users access the data
  - There can be many different views, e.g., students who takes SDSC5003 and is older than 20 years old.



Data Independence

- Physical data independence
  - Allows the physical schema to be modified without rewriting application programs
    - e.g. adding or removing an index or moving a file to a different disk
- Logical data independence
  - Allows the logical schema to be modified without rewriting application programs
    - e.g. adding an attribute to a relation
- This results in reduced application development and maintenance time

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)*

Efficient Access

- What happens when a user wants to find one record out of millions?
- An index structure maps the desired attribute values to the address of the record
- The desired records can be retrieved without scanning the whole relation

Concurrency Control

- What happens if two users try to change the same record at the same time?
- A DBMS system ensures that concurrent transactions leave the DB in a consistent state
  - While still allowing for maximal possible access of the data
    - e.g. many users can read the same record at the same time but only one user at a time can modify a record

Data Integrity

- Data should be consistent with the information that it is modeling
- A DBMS cannot actually understand what data represents
  - Users can specify integrity constraints on data and a DBMS will then enforce these constraints
    - e.g. not allowing ages to be negative

Transactions

- Changes to a DB occur as a result of *transactions*
- A transaction is a sequence of reads and writes to the DB caused by one execution of a user program
- Transactions must have the **ACID** properties:
  - **A**tomic: all or nothing
  - **C**onsistent: the DB must be in a consistent state after the transaction
  - **I**solated: transactions are performed serially
  - **D**urable: the effects of a transaction are permanent
- Log transactions for redo or undo (crash recovery)

Database Languages

Database Languages

- A database language is divided into two parts
  - Data definition language (DDL)
  - Data manipulation language (DML)
- Structured query language (SQL) is both a DDL and a DML
  - Most commercial databases use SQL and we will cover it in detail in this course

Data Definition Language

- The DDL allows entire databases to be created, and allows integrity constraints to be specified
  - Domain constraints
  - Referential integrity
  - Assertions
  - Authorization
- The DDL is also used to modify existing DB schema
  - Addition of new tables
  - Deletion of tables
  - Addition of attributes

Data Manipulation Language

- The DML allows users to access or change data in a database
  - Retrieve information stored in the database
  - Insert new information into database
  - Delete information from the database
  - Modify information stored in the database

Database Users

Database Users

- End users
  - May have specialized knowledge (CAD etc.) and may be familiar with SQL
  - The majority have no DB knowledge
- DB Administrators
  - Have central control over data and programs that access that data
- Database Application Programmers
  - Write programs that need to interact with the DB
- DB Implementers and Vendors
  - Build and sell DB products

SDSC5003

SDSC5003 Topics

- DB specification and implementation
  - Database design – the relational model and the ER model
  - Creating and accessing a database
    - Relational algebra
    - Creating and querying a DB using SQL
    - Query optimization, transaction processing, …
  - *Database application development*
  - Spark/Hadoop (more of data processing systems)
    - Solution to parallel massive data manipulation
    - Basic programming model

Textbook

- **[RG]** ***Database Management Systems (3rdedition)***, Raghu Ramakrishnan and Johannes Gehrke.
- [GUW] *Database Systems: The Complete Book (2ndedition)*, Hector Garcia-Molina, Jeffrey Ullman, and Jennifer Widom.

Grading

- Assignments:30% (10%*3)
- Midterm: 20%
- Final Exam: 30%
- Project: 20%
- 4 or 5 students per team
- Team up with people of different backgrounds!

Exercise

Which of the following plays an important role in *representing* information about the real world in a database?

1. The data definition language.
2. The data manipulation language.
3. The buffer manager.
4. The data model.

Database Design

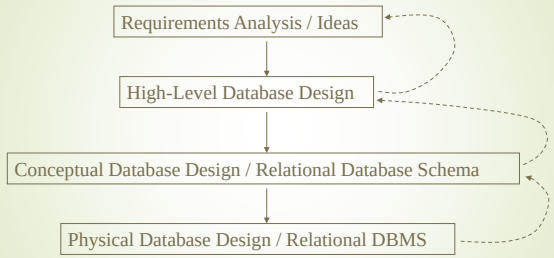
Conceptual Database Design

- *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.

SDSC5003
The Entity Relationship Model

Instructor: Yu Yang
yuyang@cityu.edu.hk

Design Overview



→ Similar to software development

Entities

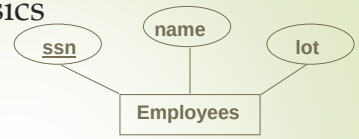
Overview

- Database design
- The ER model
  - Key Constraints.
  - Participation Constraints.
  - Weak Entities.
  - ISA Hierarchy.

Requirements Analysis

- To model a real-world enterprise
- What data are to be stored in the database?
- What applications are required to work with the database?
- Which are the most frequent, and the most important operations?

ER Model Basics



- *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* (A set of all possible attribute values, no value indicated by NULL).

Keys

- Differences between entities in an entity set are expressed in terms of their attributes
  - That is, you cannot have two different entities that have the same values for each of their attributes
    - As they would represent the same real-world entity (but see weak entity sets later)
- A *key* is a set of attributes whose values ***uniquely*** identify an entity **in an entity set**

Relationships

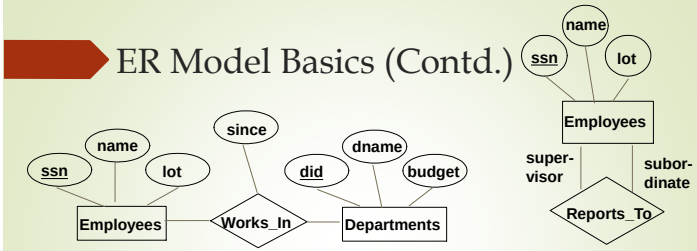
Exercise

- Let A = {1,2,3}, B = {x,y}.
- Compute B x A.
- Compute A x A.

Types of Keys

- Superkey
  - Any set of attributes whose values uniquely identify an entity in an entity set
- Candidate key
  - A minimal superkey, that is a superkey with no *extraneous* attributes
  - A relation can have more than one candidate key
- Primary key (must be a candidate key)
  - The candidate key designated by the database designer to refer to tuples (rows) in the relation (table)
  - Should never (or very rarely) change

ER Model Basics (Contd.)



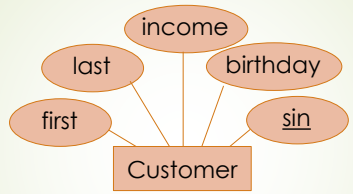
- *Relationship*: Association among two or more entities. E.g., Kris works in Pharmacy department.
- *Relationship Set*: Collection of similar relationships.
  - An n-ary relationship set R relates n entity sets E_1 ... E_n; each relationship in R involves entities e_1, ..., e_n.
    - Same entity set could participate in different relationship sets, or in different "roles" in same set.
    - It may have descriptive attributes

N-fold cross products

- Given n sets A1, A2, …, A_n, the cross product A1 x A2 x A3 …x A_n is a new set defined by

  A1 x A2 x A3 …x A_n =
  {<a1, a2, a3, …, a_n>: a1 in A1, a2 in A2, …, a_n in A_n}.
- Example: {1,2,3} x {x,y} x {1,2,3} has as members <1,x,1> and <1,y,3>.

Primary Key



Candidate Key 1: {first, last, birthday}*

Candidate Key 2: {sin}

Primary Key: {sin}

*assuming (unrealistically) that there are no two people with the same first name, last name and birthday

If {sin} is the only candidate key, how many superkeys do we have?

Cartesian or Cross-Products

- A **tuple** <a_1,a_2,…,a_n> is just a list with n elements in order.
- A binary tuple <a,b> is called an **ordered pair**.
- Given two sets A,B, we can form a new set A x B containing all ordered pairs <a,b> such that a is a member of A, b is a member of B.
- In set notation: A x B = {<a,b> | a in A, b in B}.
- Example: {1,2,3} x {x,y} = {<1,x>,<1,y>,<2,x>,<2,y>,<3,x>,<3,y>}

Exercise

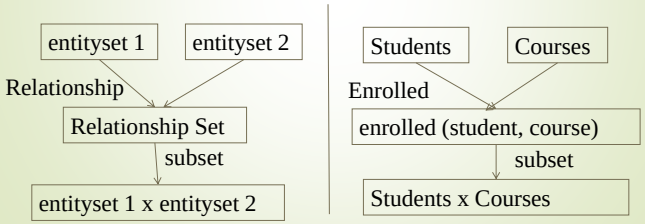
- Let A = {1,2,3}, B = {x,y}.
- Compute B x A x B.
- Compute B x B x B.
- If a set C has *n* members, how many are there in C x C? How many in C x C x C? In general, how many in C x C x …x C where we take *k* cross-products?

The Cross Product

- Let E1, E2, E3 be three entity sets.
- A **relationship** among E1, E2, E3 is a tuple in E1 x E2 x E3.
- A **relationship set** is a set of relationships. So if R is a relationship set among E1, E2, E3, then R is a *subset* of E1 x E2 x E3.
- *Relationship set = subset of cross product.*

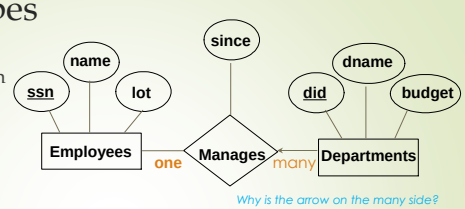
Relationships

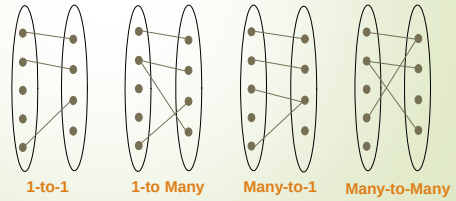
- A relationship, or simply relation, returns a relationship set given entity sets.
- Informally, the relationship specifies which entities are related.



Relation Types

- Consider Works_In: An employee can work in many departments; a dept can have many employees.
- In contrast, each dept has at most one manager, according to the *key constraint* on Manages.
- Key constraint is represented by an arrow →





Relationship Set Primary Keys

- The primary key of a relationship depends on the key constraints in the relationship
  - *Many-to-many* – all the non-descriptive attributes of the relationship set
  - *One-to-many* – the primary key for the *many* entity
  - *One-to-one* – the primary key of *either* entity

Exercise 2.2

A university database contains information about professors (identified by SSN) and courses (identified by courseid). Professors teach courses; each of the following situations concerns the Teaches relationship set. For each diagram, draw an ER diagram that describes it (assuming no further constraints hold).

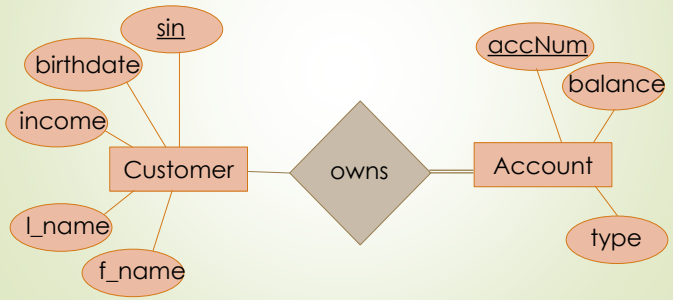
1. Professors can teach the same course in several semesters, and each offering must be recorded.
2. Professors can teach the same course in several semesters, and only the most recent such offering needs to be recorded.

Participation Constraints

- Indicate that each entity in an entity set *must* be involved in at least one relationship
- Participation is said to be either *total* (there is a constraint) or *partial* (no constraint)
  - If there is no participation constraint all the entities *may* still be involved in the relationship
- Total participation is indicated by a double line from the relationship to the entity
  - Or a thick line

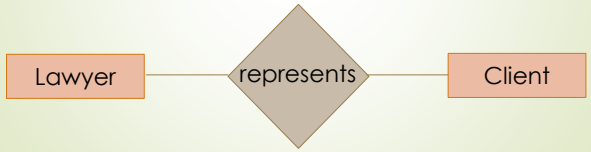
Participation Example

- Each account must be owned by at least one customer



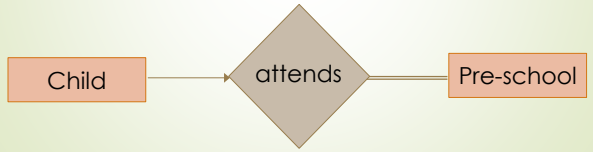
Lawyers

- A law firm's lawyers are assigned to represent clients' interests



Kids

- Children attend pre-school, assume that
  - Children can only attend one pre-school
  - A pre-school must have children



Wage Slaves

- Employees work for companies
  - An employee must be employed by someone (or they wouldn't be an employee)
  - People often have more than one job

The Man

- Chief Executive Officer (CEOs) of a company
  - Assume that being a CEO is a full time position and
  - That a company can only have one CEO
  - And that one woman or man must be in charge
    - So no cooperatives or workers' collectives

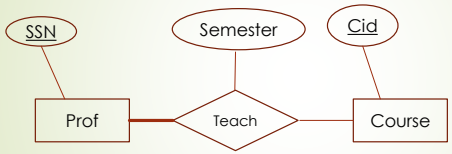
Exercise 2.2 ctd.

Same scenario as before, where we need only the current semester. Draw E-R diagrams for the following constraints.

3. Every professor must teach some course.
4. Every professor teaches exactly one course.
5. Every professor teaches exactly one course, and every course must be taught by some professor.

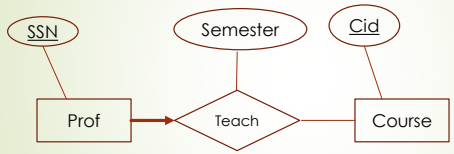
Exercise 2.2

Every professor must teach some course.



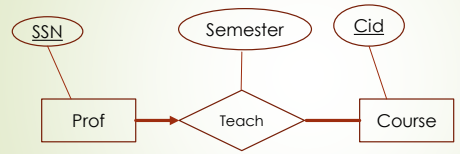
Exercise 2.2

Every professor teaches exactly one course.



Exercise 2.2

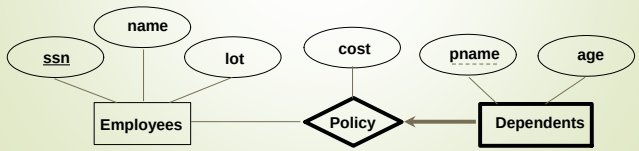
Every professor teaches exactly one course, and every course must be taught by some professor.



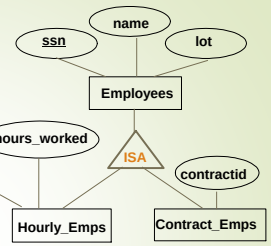
Extended ER Model

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.



ISA ('is a') Hierarchies



If we declare A **ISA** B, every A entity is also considered to be a B entity.

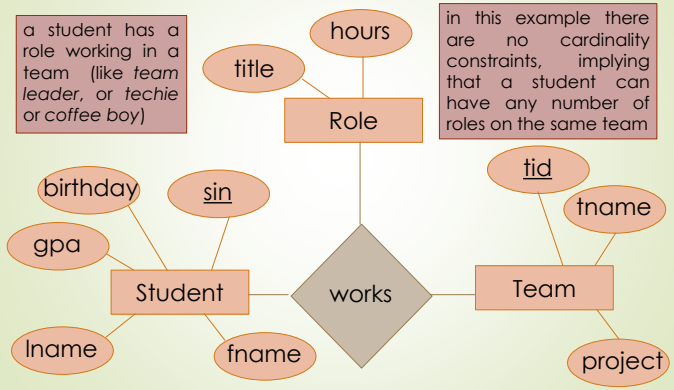
- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? (*Allowed/disallowed*)
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? (*Yes/no*)
- Reasons for using ISA:
  - To add descriptive attributes specific to a subclass.
  - To identify entities that participate in a relationship.

Non-Binary Relationships

- A relationship is does not have to be binary but can include any number of entity sets
  - Ternary relationships are not uncommon
- Specifying the mapping cardinalities of non-binary relationships can be complicated
- A key constraint indicates that an entity can participate in only one relationship
  - Just like with binary relationships

Ternary Relationships

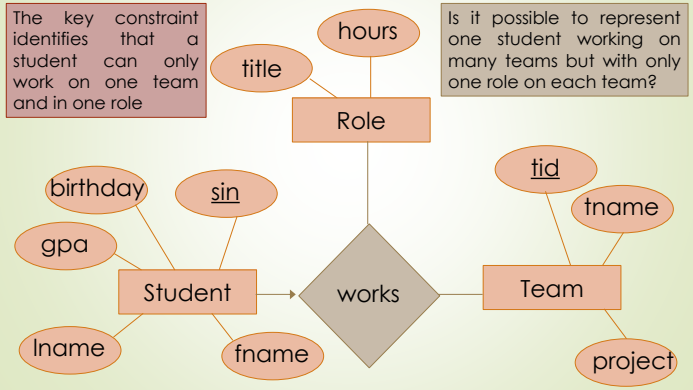
a student has a role working in a team (like *team leader*, or *techie* or *coffee boy*)



in this example there are no cardinality constraints, implying that a student can have any number of roles on the same team

Ternary Relationships

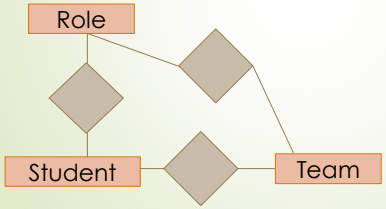
The key constraint identifies that a student can only work on one team and in one role



Is it possible to represent one student working on many teams but with only one role on each team?

Replace Ternary Relationship

- We can replace the ternary relationships with binary relationships
  - But can have only one role on each team?
  - And the relationship pairs are not forced to relate to each other
    - Bob may have a manager role and Bob may work on the Harmony OS project with a team
    - But the Harmony OS project may not have a manager



(Student,Role,Team)-> (Student,Role) + (Student,Team) + (Role,Team)

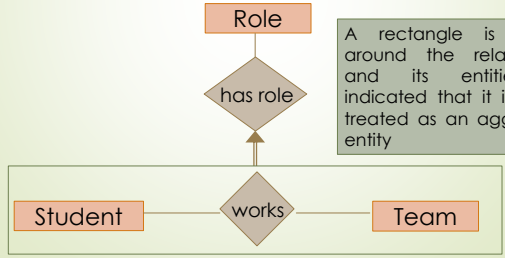
Do (Bob,Manager) + (Bob,Harmony) + (Manager,Harmony) imply (Bob,Manager,Hamony)?

Aggregation

- Indicates that a relationship set participates in another relationship set
  - An abstraction that treats relationship sets as higher-level entities
  - For the purpose of participation in other relationships
- When should aggregation be used?
  - When there is a relationship between an entity set and another relationship
  - Aggregation is often used with non-binary relationships

Aggregation Example

- Treat the works relationship as an aggregate entity
  - Each works relationship can be associated with just one role
    - Different works relationships with the same project or employee can be associated with other roles



A rectangle is drawn around the relationship and its entities to indicated that it is being treated as an aggregate entity

ER Design Principles

ER Design Principles

- Faithfulness
  - The design must be faithful to the specification (and the enterprise that is being represented)
  - All the relevant aspects of the enterprise should be represented in the model
- Avoid redundancy
  - Redundant representation makes the ER diagram harder to understand
  - Redundancy wastes storage in the DB and
  - May lead to inconsistencies

More ER Design Principles

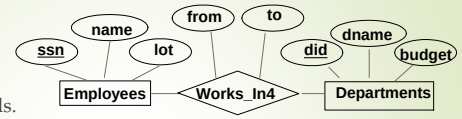
- Simplicity
  - The simpler it is, the easier it is to understand
  - Avoid introducing unnecessary entities or relationships
  - Where possible use attributes rather than entity sets or relationships
- Specify as many constraints as possible
  - Ensure that all key constraints and participation constraints are specified
  - Some constraints cannot be shown in ER diagrams
    - Example: if a team has more than 10 members, it should have one manager.

Design Choices

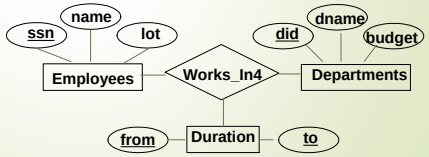
- Entity set or attribute?
- Entity set or relationship set?
- What sort of relationship is it?
  - Multiple binary relationships
  - Ternary relationship
  - Aggregation

Entity vs. Attribute

- Works_In4 does not allow an employee to work in a department for two or more periods.

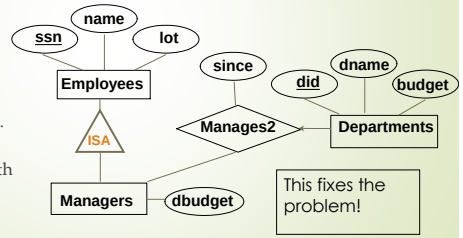


- Similar to the problem of wanting to record several addresses for an employee: We want to record *several values of the descriptive attributes for each instance of this relationship.* Accomplished by introducing new entity set, Duration.



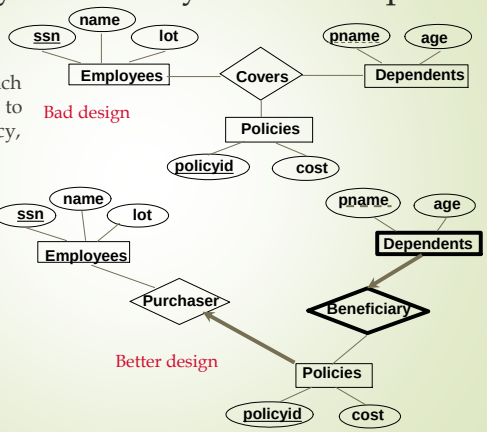
Entity vs. Relationship

- First ER diagram OK if a manager gets a separate discretionary budget for each dept.
- What if a manager gets a discretionary budget that covers *all* managed depts?
  - Redundancy: *dbudget* stored for each dept managed by manager.
  - Misleading: Suggests *dbudget* associated with department-mgr combination.



Binary vs. Ternary Relationships

- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints in the 2nd diagram?



Summary

Summary of Conceptual Design

- *Conceptual design* follows *requirements analysis*,
  - Yields a high-level description of data to be stored
- ER model popular for conceptual design
  - Constructs are expressive, close to the way people think about their applications.
- Basic constructs: *entities*, *relationships*, and *attributes* (of entities and relationships).
- Some additional constructs: *weak entities*, *ISA hierarchies*.
- Note: There are many variations on ER model.

Summary of ER (Contd.)

- Several kinds of integrity constraints can be expressed in the ER model: *key constraints*, *participation constraints*, and *overlap/covering constraints* for ISA hierarchies.
  - Some constraints (notably, *functional dependencies*) cannot be expressed in the ER model. (e.g., z = x + y)
  - Constraints play an important role in determining the best database design for an enterprise.

Summary of ER (Contd.)

- ER design is *subjective*. There are often many ways to model a given scenario! Analyzing alternatives can be tricky, especially for a large enterprise. Common choices include:
  - Entity vs. attribute.
  - entity vs. relationship
  - binary or Ternary relationship
  - ISA hierarchies.
- Ensuring good database design: resulting relational schema should be analyzed and refined further. See Ch. 19

Readings

- **RG Chapter 2**
- GUW Chapter 4.1-4.6

SDSC5003

The Relational Model

Instructor: Yu Yang
yuyang@cityu.edu.hk

Relational Database: Definitions

- *Relational database:* a set of *relations*
- *Relation = Instance + Schema*
  - *Instance* : a *table*, with rows and columns.
    #Rows = *cardinality*, #fields = *degree or 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* (all rows are distinct).

SQL

Overview

- The Relational Model
- Creating Relations in SQL

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?

The SQL Query Language

- Developed by IBM (system R) in the 1970s
- Need for a standard since it is used by many vendors
- Standards:
  - SQL-86
  - SQL-89 (minor revision)
  - SQL-92 (major revision)
  - SQL-99 (major extensions)
  - ...SQL-2011

Relational Model

Quick Question

How many distinct tuples are in a relation instance with cardinality 22?

The SQL Query Language: Preview

- To find all 18-year old students, we can write:

```
SELECT *
FROM Students S
WHERE S.age=18
```

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

- To find just names and logins, replace the first line

```
SELECT S.name, S.login
```

Exercise

1. Modify this query so that only the login column is included in the answer.
2. If the clause WHERE S.gpa >= 3.3 is added to the original query, what is the set of tuples in the answer?
3. What if the clause WHERE S.gpa > Jones is added?

```
SELECT *
FROM Students S
WHERE S.age=18
```

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

Querying Multiple Relations

- What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

Given the following instance of Enrolled:

sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53688	Topology112	A
53666	History105	B

we get:

S.name	E.cid
Smith	Topology112

Creating Tables in SQL

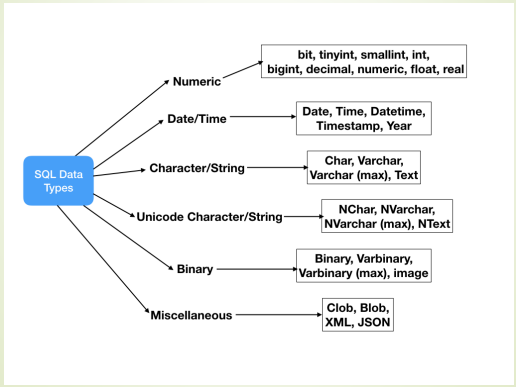
Creating Tables

- Creates the Students relation.
  - Specify the table name and field names
  - The type (domain) of each field is specified.
- The Enrolled table holds information about courses that students take.

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

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

SQL Data Types



Adding and Deleting Tuples

- Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

- Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```

- Delete all records

```
DELETE
FROM Students
```

Destroying and Altering Relations

```
DROP TABLE Students
```

- Destroys the relation Students.
- Delete not only the records, but also the schema information.

```
ALTER TABLE Students
ADD year INTEGER
```

- Add a column

```
ALTER TABLE Students
Drop year
```

- Delete a column

Modifying Records

- Note that the **WHERE** statement is evaluated *before* the **SET** statement
- An **UPDATE** statement may affect more than one record

```
UPDATE Customer
SET age = 37
WHERE sin = '111'
```

Specifying Constraints in SQL

Integrity Constraints (ICs)

- An integrity constraint restricts the data that can be stored in a DB
  - To prevent invalid data being added to the DB
    - e.g. two people with the same SIN or
    - someone with a negative age
- When a DB schema is defined, the associated integrity constraints should also be specified
- A DBMS checks every update to ensure that it does not violate any integrity constraints
  - A *legal* instance of a relation is one that satisfies all specified ICs.

Types of Integrity Constraints

- Domain Constraints
  - Specified when tables are created by selecting the type of the data
    - e.g. `age INTEGER`
- Key Constraints
  - Identifies primary keys and other candidate keys
  - Many to one or one to many
- Foreign Key Constraints
  - References primary keys of other tables
- General constraints

Primary Key Constraints

- A set of fields is a (candidate) *key* for a relation if :
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key.
  - Part 2 false? A *superkey*.
  - If there is >1 key for a relation, one of the keys is chosen to be the *primary key*.
- Examples.
  - *sid* is a key for Students. (What about *name*?)
  - The set {*sid, gpa*} is a superkey.

Primary and Candidate Keys in SQL

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

- "For a given student and course, there is a single grade." vs. "Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade."
- What does the Unique constraint do? Is this a good idea?

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid) )
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid),
   UNIQUE (cid, grade) )
```

Exercise

- Assume that a patient can be uniquely identified by either SIN or MSP number
  - SIN is chosen as the primary key

Exercise

- Assume that a patient can be uniquely identified by either SIN or MSP number
  - SIN is chosen as the primary key

```
CREATE TABLE Patient (
   sin      CHAR(11),
   msp      CHAR(15),
   fName    CHAR(20),
   lName    CHAR(20),
   age      INTEGER,
   UNIQUE (msp)
   PRIMARY KEY (sin) )
```

Exercise (cont.)

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

1. Give an example of an attribute (or set of attributes) that you can deduce is *not* a candidate key, if this instance is legal.
2. Is there any example of an attribute (or set of attributes) that you can deduce *is* a candidate key?
3. Does every relational schema have *some* candidate key?

Foreign Keys, Referential Integrity

- *Foreign key* : Set of fields in one relation that is used to refer to a tuple in another relation.
- Must correspond to primary key in another relation.
- Like a pointer.
- E.g. *sid* is a foreign key referring to Students:
  - Enrolled(*sid*: string, *cid*: string, *grade*: string)
  - If all foreign key constraints are enforced, *referential integrity* is achieved, i.e., no dangling references.

Foreign Keys in SQL

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

```
CREATE TABLE Enrolled
  (sid CHAR(20), cid CHAR(20), grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid) REFERENCES Students )
```

Enrolled

sid	cid	grade
53666	Carnatic101	C
53666	Reggae203	B
53650	Topology112	A
53666	History105	B

Students

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

Enforcing Referential Integrity

- *sid* in Enrolled is a foreign key that references Students.

1. What should be done if an Enrolled tuple with a non-existent student id is inserted?
2. What should be done if a Students tuple is deleted, e.g. sid = 53666?
   - A. Delete all Enrolled tuples that refer to 53666.
   - B. Disallow deletion 53666.
   - C. Set sid in Enrolled tuples that refer to 53666 to a *default sid*.
   - D. Set sid in Enrolled tuples that refer to it to a special value *null*, denoting '*unknown*' or '*inapplicable*'.

Foreign Keys – Insertions in the Referencing Table

accnum	balance	type	sin
761	904.33	CHQ	111
856	1011.45	CHQ	333
903	12.05	CHQ	222
1042	10000.00	SAV	333

Inserting {409, 0, CHQ, 555} into *Account* violates the foreign key on *sin* as there is no *sin* of 555 in *Customer*

sin	fname	lname	age	salary
111	Buffy	Summers	23	43000.00
222	Xander	Harris	22	6764.87
333	Rupert	Giles	47	71098.65
444	Dawn	Summers	17	4033.32

Foreign Keys – Insertions in the Referencing Table

accnum	balance	type	sin
761	904.33	CHQ	111
856	1011.45	CHQ	333
903	12.05	CHQ	222
1042	10000.00	SAV	333

Inserting {409, 0, CHQ, 555} into *Account* violates the foreign key on *sin* as there is no *sin* of 555 in *Customer*

The insertion is rejected; before it is processed a *Customer* with a *sin* of 555 must be inserted into the *Customer* table

sin	fname	lname	age	salary
111	Buffy	Summers	23	43000.00
222	Xander	Harris	22	6764.87
333	Rupert	Giles	47	71098.65
444	Dawn	Summers	17	4033.32

Foreign Keys – Updates to the Referencing Table

accnum	balance	type	sin
761	904.33	CHQ	111
856	1011.45	CHQ	333
903	12.05	CHQ	222
1042	10000.00	SAV	333

Changing this record's *sin* to 555 also violates the foreign key, again leading to the transaction being rejected

sin	fname	lname	age	salary
111	Buffy	Summers	23	43000.00
222	Xander	Harris	22	6764.87
333	Rupert	Giles	47	71098.65
444	Dawn	Summers	17	4033.32

Foreign Keys – Deletions in the Referenced Table

accnum	balance	type	sin
761	904.33	CHQ	111
856	1011.45	CHQ	333
903	12.05	CHQ	222
1042	10000.00	SAV	333

Deleting this record will violate the foreign key, because a record with that *sin* exists in the *Account* table

sin	fname	lname	age	salary
111	Buffy	Summers	23	43000.00
222	Xander	Harris	22	6764.87
333	Rupert	Giles	47	71098.65
444	Dawn	Summers	17	4033.32

Foreign Keys – Updates to the Referenced Table

accnum	balance	type	sin
761	904.33	CHQ	111
856	1011.45	CHQ	333
903	12.05	CHQ	222
1042	10000.00	SAV	333

Updating this record so that the *sin* = 666 will violate the foreign key, because a record with the original *sin* exists in the *Account* table

sin	fname	lname	age	salary
111	Buffy	Summers	23	43000.00
222	Xander	Harris	22	6764.87
333	Rupert	Giles	47	71098.65
444	Dawn	Summers	17	4033.32

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)

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid),
   FOREIGN KEY (sid)
     REFERENCES Students
     ON DELETE CASCADE
     ON UPDATE SET DEFAULT )
```

General Constraints

- A DB may require constraints other than primary keys, foreign keys and domain constraints
  - Limiting domain values to subsets of the domain
    - e.g. limit age to positive values less than 150
  - Or other constraints involving multiple attributes
- SQL supports two kinds of general constraint
  - *Table constraints* associated with a single table
  - *Assertions* which may involve several tables and are checked when any of these tables are modified
- These will be covered later in the course

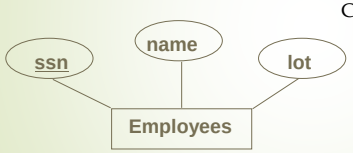
Translate ER Diagrams to SQL

Problem Solving Steps

- Understand the business rules/requirements
- Draw the ER diagram
- Draw the Relational Model
- Write the SQL and create the database

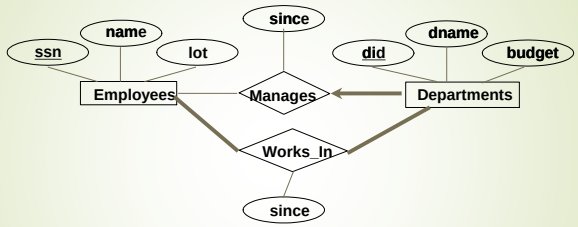
Logical DB Design: ER to Relational

- Entity sets to tables:




```
CREATE TABLE Employees
  (ssn CHAR(11),
  name CHAR(20),
  lot  INTEGER,
  PRIMARY KEY  (ssn))
```

Review: The Works_In Relation



Exercise:

1. Write a create statement for the Departments entity set.
2. Write a create statement for the Works_In relation. Do not consider the participation constraint now.

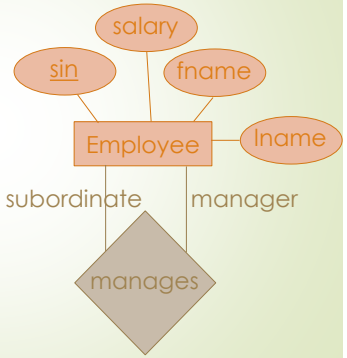
Relationship Sets to Tables

- 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 *key* for the relation. (Superkey?)
  - All descriptive attributes.

```
CREATE TABLE Works_In(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY (ssn, did),
  FOREIGN KEY (ssn)
     REFERENCES Employees,
  FOREIGN KEY (did)
     REFERENCES Departments)
```

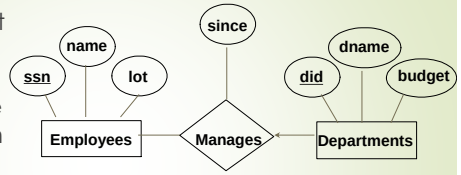
Relationship Set to Table …

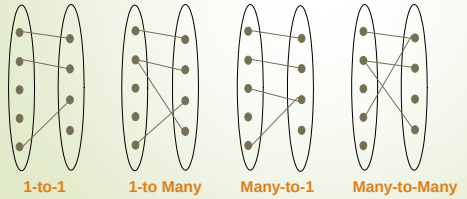
```
CREATE TABLE Manages  (
   manSINCHAR(11),
   subSIN CHAR(11),
   FOREIGN KEY (manSIN)
   REFERENCES Employee,
   FOREIGN KEY (subSIN)
   REFERENCES Employee,
   PRIMARY KEY (manSIN,
   subSIN) )
```



Key Constraints

- Each dept has at only one manager, according to the *key constraint* on Manages.





Translation to relational model?

Translating ER Diagrams with Key Constraints

- Map relationship to a table:
  - Note that did is the key now!
  - Separate tables for Employees and Departments.

```
CREATE TABLE  Manages(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees,
  FOREIGN KEY (did) REFERENCES Departments)
```

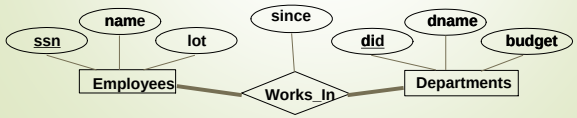
- Since each department has a unique manager, we could instead combine Manages and Departments.

```
CREATE TABLE  Dept_Mgr(
  did  INTEGER,
  dname  CHAR(20),
  budget  REAL,
  ssn  CHAR(11),
  since  DATE,
  PRIMARY KEY  (did),
  FOREIGN KEY (ssn) REFERENCES Employees)
```

Participation Constraints

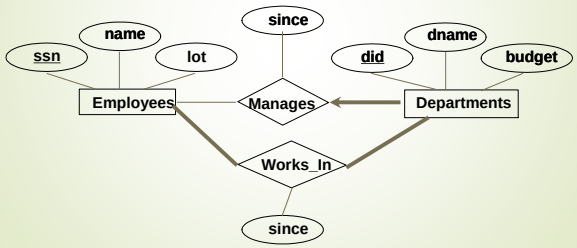
- Every department must have some employee.
- Each employee must work in some department.
- Can we capture these constraints?

```
CREATE TABLE Works_In(
  ssn  CHAR(11),
  did  INTEGER,
  since  DATE,
  PRIMARY KEY (ssn, did),
  FOREIGN KEY (ssn)
     REFERENCES Employees,
  FOREIGN KEY (did)
     REFERENCES Departments)
```



Participation Constraint + Key Constraint

- Every department must have a manager.
- Can we capture this constraint?

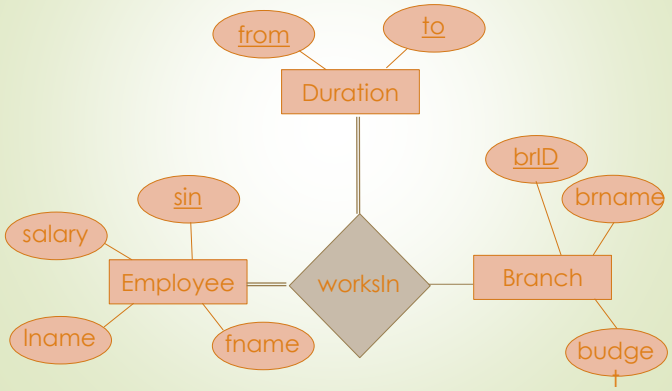


Participation Constraints in SQL

- We can capture participation constraints involving one entity set in a binary relationship.
- But little else (with what we have so far).

```
CREATE TABLE Dept_Mgr(
    did INTEGER,
    dname CHAR(20),
    budget REAL,
    ssn CHAR(11) NOT NULL,
    since DATE,
    PRIMARY KEY (did),
    FOREIGN KEY (ssn) REFERENCES Employees,
      ON DELETE NO ACTION)
```

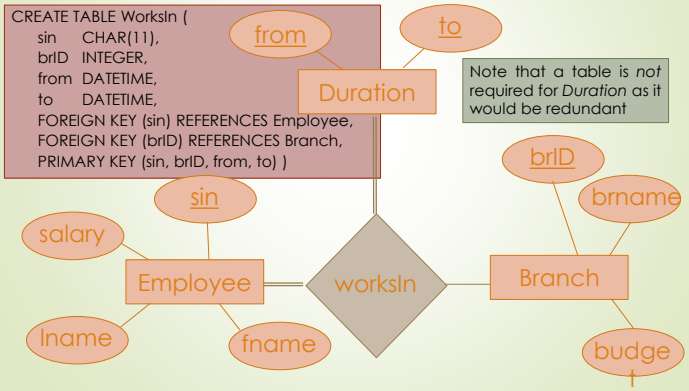
More Relationship Sets …



More Relationship Sets …

```
CREATE TABLE WorksIn (
    sin     CHAR(11),
    brID    INTEGER,
    from    DATETIME,
    to      DATETIME,
    FOREIGN KEY (sin) REFERENCES Employee,
    FOREIGN KEY (brID) REFERENCES Branch,
    PRIMARY KEY (sin, brID, from, to) )
```

Note that a table is *not* required for *Duration* as it would be redundant

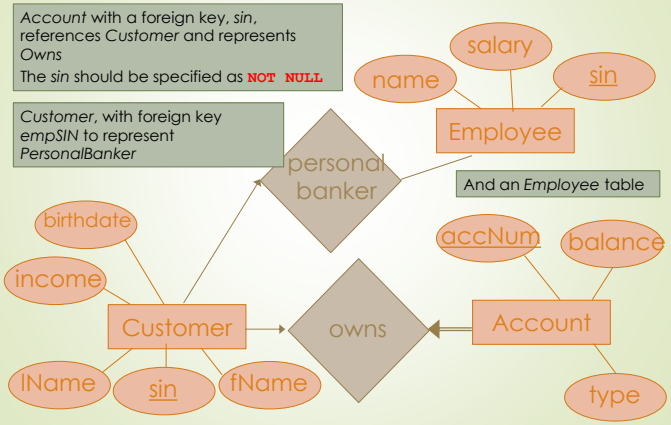


More Relationship Sets …

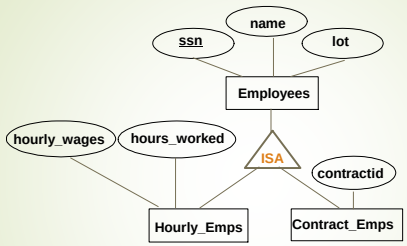
Account with a foreign key, *sin*, references *Customer* and represents *Owns*
The *sin* should be specified as `NOT NULL`

Customer, with foreign key *empSIN* to represent *PersonalBanker*

And an *Employee* table



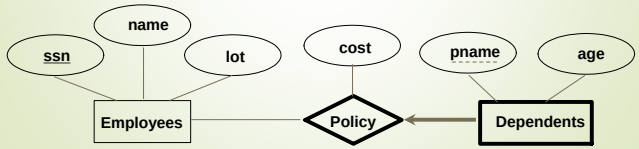
Review: ISA Hierarchies



- *Overlap constraints*: Can Joe be an Hourly_Emps as well as a Contract_Emps entity? *(Allowed/disallowed)*
- *Covering constraints*: Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? *(Yes/no)*

Review: 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 (1 owner, many weak entities).
  - Weak entity set must have total participation in this *identifying* relationship set.



Translating ISA Hierarchies to Relations

- 3 relations: Employees, Hourly_Emps and Contract_Emps.
  - Every employee is recorded in Employees. For hourly emps, extra info recorded in Hourly_Emps (*hourly_wages, hours_worked, ssn*)
- 2 relations: Just Hourly_Emps and Contract_Emps.
  - *Hourly_Emps*: *ssn, name, lot, hourly_wages, hours_worked.*
  - Each employee must be in one of these two subclasses.
- 1 relation: Employees.
  - *Emps*: *ssn, name, lot, hourly_wages, hours_worked, contractid.*
  - Requires null values.

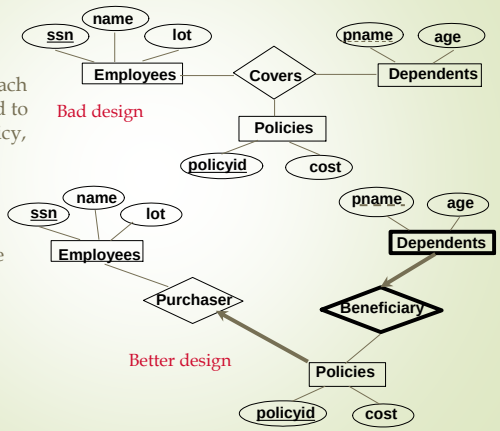
Translating Weak Entity Sets

- Weak entity set and identifying relationship set are translated into a single table.
  - When the owner entity is deleted, all owned weak entities must also be deleted.
  - What guarantees existence of owner?

```
CREATE TABLE Dep_Policy (
    pname CHAR(20),
    age INTEGER,
    cost REAL,
    owner CHAR(11),
    PRIMARY KEY (pname, owner),
    FOREIGN KEY (owner) REFERENCES Employees(ssn),
      ON DELETE CASCADE)
```

Review: Binary vs. Ternary Relationships

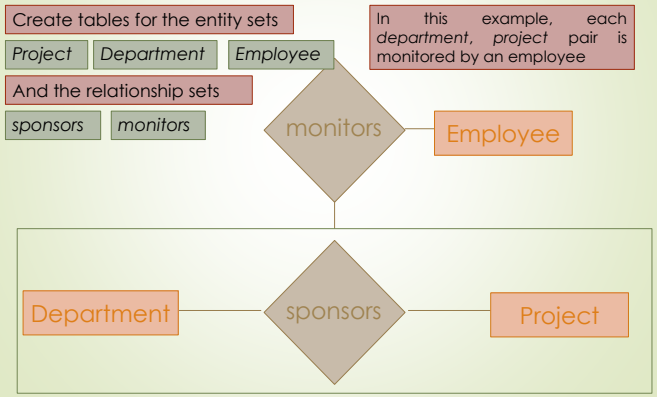
- If each policy is owned by just 1 employee, and each dependent is tied to the covering policy, first diagram is inaccurate.
- What are the additional constraints in the 2nd diagram?



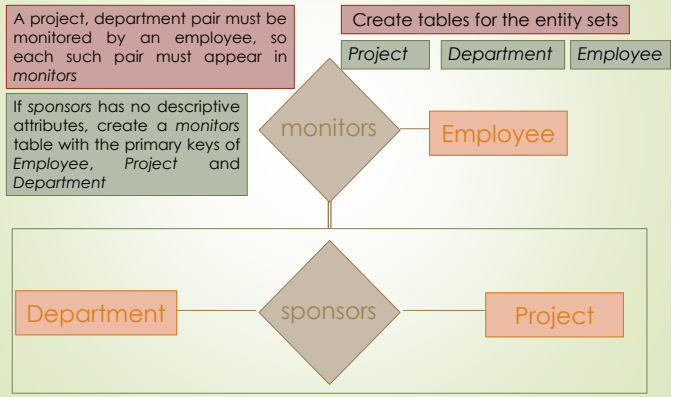
Binary vs. Ternary Relationships SQL

```
CREATE TABLE Policies (
  policyid INTEGER,
  cost REAL,
  ssn CHAR(11) NOT NULL,
  PRIMARY KEY (policyid).
  FOREIGN KEY (ssn) REFERENCES Employees,
    ON DELETE CASCADE)
CREATE TABLE Dependents (
  pname CHAR(20),
  age INTEGER,
  policyid INTEGER,
  PRIMARY KEY (pname, policyid).
  FOREIGN KEY (policyid) REFERENCES Policies,
    ON DELETE CASCADE)
```

Aggregation Example



Aggregation Example



Aggregation

- An aggregate entity is represented by the table defining the relationship set in the aggregation
- The relationship *between* the aggregate entity and the other entity has the following attributes:
  - The primary key of the participating entity set, and
  - The primary key of the relationship set that defines the aggregate entity, and
  - Its own descriptive attributes, if any
- The normal rules for determining primary keys and omitting tables apply

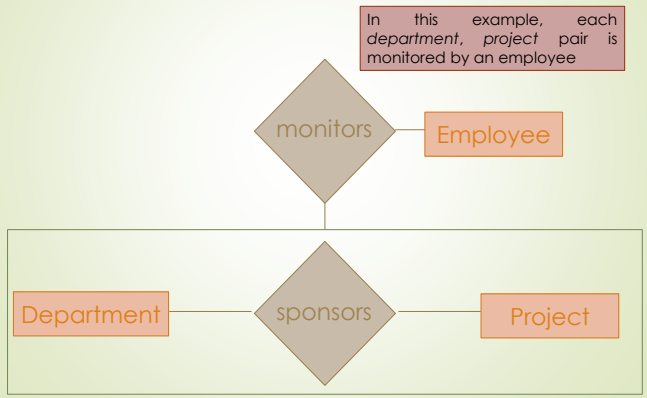
Aggregation – Special Case

- There is a case where no table is required for an aggregate entity
  - Even where there are no cardinality constraints
  - If there is *total participation* between the aggregate entity and its relationship, and
  - If the aggregate entity does not have any descriptive attributes
  - Insert the attributes of the aggregate entity into the table representing the relationship with that entity

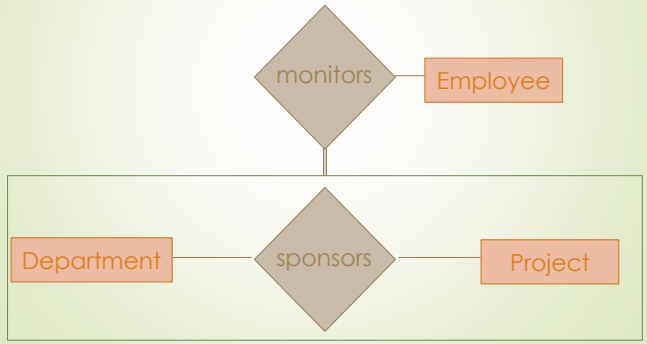
Summary: From ER to SQL

- Basic construction: each entity set becomes a table.
- Each relationship becomes a table with primary keys that are also foreign keys referencing the entities involved.
- Key constraints in ER give option of merging entity table with relationship table (e.g. Dept_Mgr).
  - Use not-null to enforce participation.

Aggregation Example



Aggregation Example



View

Introduction to Views

- Views are not explicitly stored in a DB but are created as required from a view definition
  - At least conceptually, if not necessarily in practice
- Once defined, views may be referred to in the same way as a tables
- View are useful for
  - Convenience – users can access the data they require without referring to many tables
  - Security – users can only access appropriate data
  - Independence – views can help mask changes in the conceptual schema

Exercise ctd.

How will the system process the query:

SELECT S.sname

FROM SeniorEmp S

WHERE S.salary > 100,000

Relational Query Languages

- *Query languages*: Allow manipulation and retrieval of data from a database.
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on algebra/logic.
  - Allows for much optimization.
- A query language is not a general purpose programming language
  - They are not Turing complete, and
  - Are not intended for complex calculations

Views

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

```
CREATE VIEW YoungActiveStudents (name, grade)
    AS SELECT S.name, E.grade
    FROM Students S, Enrolled E
    WHERE S.sid = E.sid and S.age<21
```

- Views can be dropped using the DROP VIEW command.

Readings

- **RG Chapter 3**
- GUW Chapter 2

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - *Relational Algebra*: More operational, very useful for representing execution plans.
  - *Relational Calculus*: Lets users describe what they want, rather than how to compute it. (Non-operational, *declarative*.) Not covered in course.
- Understanding formal query languages is important for understanding
  - The origins of SQL
  - Query processing and optimization

Exercise 3.19

Consider the following **schema**.

Emp(eid: integer, ename: string, age: integer, salary: real)

Works(eid: integer, did: integer, pct_time: integer)

Dept(did: integer, budget: real, managerid: integer)

And the **view**

CREATE VIEW SeniorEmp(sname, sage, salary)

AS SELECT E.ename, E.age, E.salary

FROM Emp E

WHERE E.age >50

SDSC5003

Relational Algebra

Instructor: Yu Yang

yuyang@cityu.edu.hk

Procedural vs. Non-procedural

- A procedural query consists of a series of operations
  - Which describe a step-by-step process for calculating the answer
  - e.g. giving precise directions on how to get somewhere
    - "…, turn left at the blue sign, drive for 1.5 miles, turn right at the cat, then left again at the red tower, …"
- A non-procedural (or declarative) query describes *what* output is desired, and not *how* to compute it
  - e.g. giving the address someone is to go to
    - "go to the HSBC bank at 15th"

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.

Algebra

- In math, algebraic operations like +, -, x, /.
- Operate on numbers: input are numbers, output are numbers.
- Can also do Boolean algebra on sets, using union, intersect, difference.
- Focus on **algebraic identities**, e.g.
  - x (y+z) = xy + xz.
- (Relational algebra lies between propositional and 1st-order logic.)

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation.
- Assume that names of fields in query results are inherited from names of fields in query input relations.

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

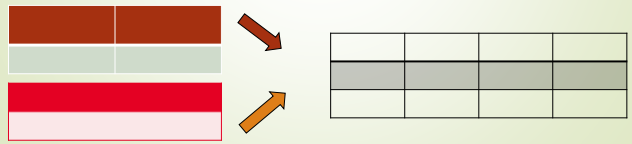
sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Preliminaries

- Fields in an instance can be referred to either by position or by name
- Positional vs. named-attribute notation:
  - Positional notation
    - Ex: Sailor(1,2,3,4)
    - easier for formal definitions
  - Named-attribute notation
    - Ex: Sailor(sid, sname, rating,age)
    - more readable

Relational Algebra

- Every operator takes one or two relation instances
- A relational algebra expression is recursively defined to be a relation
  - Result is also a relation
  - Can apply operator to
    - Relation from database
    - Relation as a result of another operator



Projection

- The projection operator, π (pi), specifies the columns to be retained from the input relation
- A selection has the form:

  $\pi_{columns}(relation)$

  - Where *columns* is a comma separated list of column names
  - The list contains the names of the columns to be retained in the result relation

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

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

age
35.0
55.5

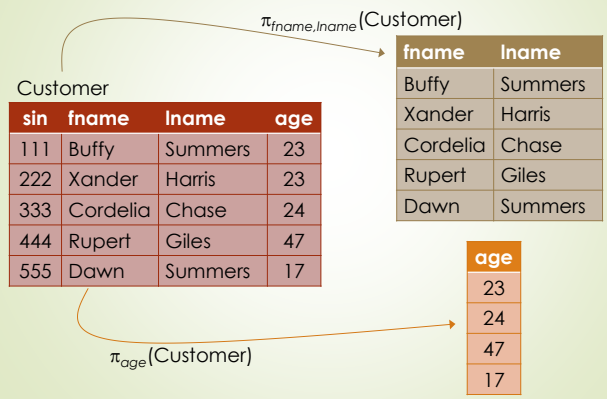
$\pi_{age}(S2)$

Relational Algebra

Relational Algebra Operations

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Selects a subset of 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 derived operations:
  - Intersection, *join*, division, renaming. Not essential, but very useful.
- Since each operation returns a relation, operations can be *composed*!

More Projection Example



Selection

- The selection operator, σ (sigma), specifies the rows to be retained from the input relation
- A selection has the form: $\sigma_{condition}(relation)$, where *condition* is a Boolean expression
  - Terms in the condition are comparisons between two fields (or a field and a constant)
  - Using one of the comparison operators: $<, \leq, =, \neq, \geq, >$
  - Terms may be connected by $\wedge$ (and), or $\vee$ (or),
  - Terms may be negated using $\neg$ (not)

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

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

sname	rating
yuppy	9
rusty	10

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

More Selection Example

$\sigma_{age > 23}$(Customer)

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

sin	fname	lname	age
333	Cordelia	Chase	24
444	Rupert	Giles	47

$\sigma_{lname = "Summers"}$(Customer)

sin	fname	lname	age
111	Buffy	Summers	23
555	Dawn	Summers	17

Selection and Projection

$\pi_{sin, fname}(\sigma_{age > 18 \wedge lname = "Summers"}$(Customer))

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

intermediate relation

sin	fname	lname	age
111	Buffy	Summers	23

sin	fname
111	Buffy

Selection and Projection Notes

- Selection and projection eliminate duplicates
  - Because relations are sets
  - In practice (SQL), duplicate elimination is expensive, therefore it is only done when explicitly requested
- Both operations require one input relation
- The schema of the result of a selection is *the same as* the schema of the input relation
- The schema of the result of a projection contains just those attributes in the projection list

Set Operations

- Relational algebra includes the standard set operations:
  - Union, ∪
  - Intersection, ∩
  - Set Difference, –
  - Cartesian product (Cross product), ×

  These are all binary operators
- All relational algebra operations can be implemented using these five basic operations:
  - *Selection*, *projection*, *union*, *set difference* and *Cartesian product*

Set Operations Review

A = {1, 3, 6}

B = {1, 2, 5, 6}

Union (∪): A ∪ B ≡ B ∪ A; A ∪ B = {1, 2, 3, 5, 6}

Intersection(∩): A ∩ B ≡ B ∩ A; A ∩ B = {1, 6}

Set Difference(–): A – B ≠ B – A; A – B = {3}; B – A = {2, 5}

Cartesian Product (×): A × B ≡ B × A*

A × B ={(1,1),(1,2),(1,5),(1,6),(3,1),(3,2),(3,5), (3,6),(6,1),(6,2),(6,5),(6,6)}

* not strictly true, as pairs are ordered

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation.
- Assume that names of fields in query results are inherited from names of fields in query input relations.

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

Union, Intersection, Set-Difference

- All 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

$S1 \cap S2$

Exercise on Union

Number	shape	holes
1	round	2
2	square	4
3	rectangle	8

Blue blocks (BB)

Number	shape	holes
4	round	2
5	square	4
6	rectangle	8

Yellow blocks(YB)

Stacked(S)

bottom	top
4	2
4	6
6	2

1. Which tables are union-compatible?
2. What is the result of the possible unions?

More Union Compatible Relations

Intersection of the Employee and Customer relations

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

Employee

sin	fname	lname	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
412	Carol	Danvers	64000.00

The two relations are not union compatible as age is an INTEGER and salary is a REAL

More Union Compatible Relations

Intersection of the Employee and Customer relations

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

Employee

sin	fname	lname	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
412	Carol	Danvers	64000.00

The two relations are not union compatible as age is an INTEGER and salary is a REAL

However we can carry out some preliminary operations to make the relations union compatible:

$\pi_{sin,\ fname,\ lname}(\text{Customer}) \cap \pi_{sin,\ fname,\ lname}(\text{Employee})$

More Union Example

R

sin	fname	lname
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	fname	lname
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

$R \cup S$

sin	fname	lname
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers
208	Clark	Kent
412	Carol	Danvers

Returns all records in either relation (or both)

More Intersection Example

R

sin	fname	lname
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	fname	lname
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

$R \cap S$

sin	fname	lname
111	Buffy	Summers

Only returns records that are in both relations

More Set Difference Example

R

sin	fname	lname
111	Buffy	Summers
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

S

sin	fname	lname
208	Clark	Kent
111	Buffy	Summers
412	Carol	Danvers

$R - S$

sin	fname	lname
222	Xander	Harris
333	Cordelia	Chase
444	Rupert	Giles
555	Dawn	Summers

R – S returns all records in R that are not in S

$S - R$

sin	fname	lname
208	Clark	Kent
412	Carol	Danvers

Cartesian (Cross) Product

- The *schema* of the result of $R \times S$ has one attribute for each attribute of the input relations
  - All of the fields of R, followed by all of the fields of S
  - Names are inherited if possible (i.e. if not duplicated)
    - If two field names are the same, a *naming conflict* occurs and the affected columns are referred to by position
- The result *relation* contains one record for each pair of records $r \in R, s \in S$
  - Each record in R is paired with each record in S
    - If R contains m records, and S contains n records, the result relation will contain $m * n$ records

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation.
- Assume that names of fields in query results are inherited from names of fields in query input relations.

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

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

Renaming

- It is sometimes useful to assign names to the results of a relational algebra query
- The rename operator, ρ (rho) allows a relational algebra expression to be renamed
  - $\rho_x(E)$ names the result of the expression, or
  - $\rho_{x(A1,A2,\ldots,An)}(E)$ names the result of the expression, and its attributes
- For example, find the account with the largest balance

Largest Balance

- To find the largest balance first find accounts which are less than some other account
  - By performing a comparison on the Cartesian product of the account table with itself
    - The *Account* relation is referred to twice so with no renaming we have an ambiguous expression:
    - $\sigma_{account.balance < account.balance}$ (Account × Account)
  - So rename one version of the Account relation
    - $\sigma_{account.balance < d.balance}$ (Account × ρ_d (Account))
- Then use set difference to find the largest balance

Example Instances

- "Sailors" and "Reserves" relations for our examples.
- We' ll use positional or named field notation.
- Assume that names of fields in query results are inherited from names of fields in query input relations.

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

Joins (cont.)

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

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

$$S1 \bowtie_{R.sid = S.sid} R1$$

- *Result schema* similar to cross-product, but only one copy of fields for which equality is specified.
- *Natural Join*: Equi-join on *all* common fields.
  Without specified condition
  means the natural join of A and B.

$$A \bowtie B$$

Exercise on Cross-Product

Number	shape	holes
1	round	2
2	square	4
3	rectangle	8

Blue blocks (BB)

Number	shape	holes
4	round	2
5	square	4
6	rectangle	8

Stacked(S)

bottom	top
4	2
4	6
6	2

1. Write down 2 tuples in BB x S.
2. What is the cardinality of BB x S?

Joins (cont.)

- *Condition Join*: $R \bowtie_c S = \sigma_c (R \times S)$

(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. How?
- Sometimes called a *theta-join*.

Example for Natural Join

Number	shape	holes
1	round	2
2	square	4
3	rectangle	8

Blue blocks (BB)

shape	holes
round	2
square	4
rectangle	8

Yellow blocks(YB)

What is the natural join of BB and YB?

Joins

- It is often useful to simplify some queries that require a Cartesian product
- There is often a natural way to join two relations
  - e.g., finding data about customers and their accounts
    - Assume *Account* has a foreign key, that references *Customer*
  - Get the Cartesian product of *Account* and *Customer*
    - Select the tuples where the primary key of *Customer* equals the foreign key attribute in *Account*

Exercise on Join

Number	shape	holes
1	round	2
2	square	4
3	rectangle	8

Blue blocks (BB)

Number	shape	holes
4	round	2
5	square	4
6	rectangle	8

Yellow blocks(YB)

$$BB \bowtie_{BB.holes < YB.holes} YB$$

Write down 2 tuples in this join.

More Natural Join Example

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

Employee

sin	fname	lname	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
396	Dawn	Allen	41000.21
412	Carol	Danvers	64000.00

Customer ⋈ Employee

sin	fname	lname	age	salary
111	Buffy	Summers	23	22000.78

More Theta Join Example

Customer

sin	fname	lname	age
111	Buffy	Summers	23
222	Xander	Harris	23
333	Cordelia	Chase	24
444	Rupert	Giles	47
555	Dawn	Summers	17

Employee

sin	fname	lname	salary
208	Clark	Kent	80000.55
111	Buffy	Summers	22000.78
412	Carol	Danvers	64000.00

Customer $\bowtie_{\text{Customer.sin} < \text{Employee.sin}}$ Employee

1	2	3	age	5	6	7	salary
111	Buffy	Summers	23	208	Clark	Kent	80000
111	Buffy	Summers	23	412	Carol	Danvers	64000
222	Xander	Harris	23	412	Carol	Danvers	64000
333	Cordelia	Chase	17	412	Carol	Danvers	64000

Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$
- Solution 2:

$$\rho\ (Temp1,\ \sigma_{bid=103} Reserves)$$

$$\rho\ (Temp2,\ Temp1 \bowtie Sailors)$$

$$\pi_{sname}\ (Temp2)$$

- Solution 3:

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

Exercise: Find names of sailors who've reserved a red boat

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

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

- A more efficient solution:

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

A query optimizer can find this, given the first solution!

Complex Exercises

- "Sailors" and "Reserves" relations for our examples.
- We'll use positional or named field notation.
- Assume that names of fields in query results are inherited from names of fields in query input relations.

Reserves

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

Sailors

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

Exercise: Find names of sailors who've reserved a red boat

Reserves

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

Sailors

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

Boats

bid	color
101	red
102	green
103	green

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

Reserves

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

Sailors

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

Boats

bid	color
101	red
102	green
103	green

Find names of sailors who've reserved boat #103

Reserves

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

Sailors

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

Exercise: Find names of sailors who've reserved a red boat

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

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

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

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

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$

$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

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

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

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}\ Boats))$$

$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

- Can also define Tempboats using union! (How?)
- What happens if $\vee$ is replaced by $\wedge$ in this query?

Exercise: Find sailors who've reserved a red and 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):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'}\ Boats) \bowtie Reserves))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'}\ Boats) \bowtie Reserves))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

Find the names of sailors who've reserved all boats

Reserves

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

Sailors

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

Boats

bid	color
101	red
102	green
103	green

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

Reserves

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

Sailors

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

Boats

bid	color
101	red
102	green
103	green

Division

- Division is useful for queries which return records associated with *all* of the records in some subset
  - Find people who like all types of music
    - Country *and* Western
  - Find tourists who have visited all of the provinces in Canada
- This operator is always implemented in DBMSs
- But it can be expressed in terms of the basic set operators
  - Implementing a division query in SQL is a *fun* exercise

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids, (\pi_{sid,bid}Reserves) / (\pi_{bid}Boats))$$

$$\pi_{sname}\ (Tempsids \bowtie Sailors)$$

Exercise: Find sailors who've reserved a red and 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):

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

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

A/B2

pno
p1
p2
p4

B3

sno
s1

A/B3

Find the names of sailors who've reserved all boats

- Uses division; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids, (\pi_{sid,bid}Reserves) / (\pi_{bid}Boats))$$

$$\pi_{sname}\ (Tempsids \bowtie Sailors)$$

- To find sailors who have reserved all red boats:

..... $/\ \rho_{bid}(s_{color='red'}\ Boats)$

Summary

- Relational algebra is a procedural language that is used as the internal representation of SQL
- There are five basic operators: selection, projection, cross-product, union and set difference
  - Additional operators are defined in terms of the basic operators: intersection, join, and division
- There are often several equivalent ways to express a relational algebra
  - These equivalencies are exploited by query optimizers to re-order the operations in a relational algebra expression (choose the most efficient one)

Introduction

- We now introduce SQL, the standard query language for relational DBS.
- Like relational algebra, an SQL query takes one or two input tables and returns one output table.
- Any RA query can also be formulated in SQL.
- In addition, SQL contains certain features that go beyond the expressiveness of RA, e.g. sorting and aggregation functions.

Conceptual Evaluation

- 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 typically the least efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Readings

- **RG Chapter 4.1,4.2**
- GUW Chapter 2.4

Simple Queries

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
1	1	Dustin	7	45	4	103	19-SEP-2017
2	2	Rusty	10	35	4	103	19-SEP-2017
3	3	Horatio	5	35	4	103	19-SEP-2017
4	4	Zorba	8	18	4	103	19-SEP-2017
5	5	Julius	NULL	25	4	103	19-SEP-2017

SDSC 5003
Structured Query Language

Instructor: Yu Yang
yuyang@cityu.edu.hk

Basic SQL Syntax

```
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 $<, >, =, \leq, \geq, \neq$) 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!

A Note on Range (Tuple) 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
```

OR

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

Tuple Variables

- Tuple (or range) variables allow tables to be referred to in a query using an alias
  - The tuple variable is declared in the **FROM** clause by noting it immediately after the table it refers to
- If columns in two tables have the same names, tuple variables *must* be used to refer to them
  - Tuple variables are *not required* if a column name is *unambiguous*
- The full name of a table is an implicit tuple variable
  - e.g. **SELECT Customer.fname** …

Why Use Tuple Variables?

- There are a number of reasons to use (short) explicit tuple variables
  - Distinguishing between columns with the same name that are from different tables
  - Readability
  - Laziness, **C.age** is less to type than **Customer.age**
  - To make it easier to re-use queries
  - Because a query refers to the *same* table twice in the **FROM** clause

Required Range Variables

- Find the names and SINs of customers who have an income greater than Rupert Giles
  - By comparing each customer's income to Rupert's income
  - Using two references to the customer table

```
SELECT C.osin, C.fname, C.lname
FROM Customer C, Customer RG
WHERE RG.fname = 'Rupert' AND RG.lname =
    'Giles' AND C.income > RG.income
```

The implicit range name, *Customer*, cannot distinguish between the two references to the *Customer* table

What would the query return if there are two customers called Rupert Giles?

π-σ-× Queries

- SELECT [DISTINCT] *S.sname*
  - π_{sname}
- FROM *Sailors, Reserves*
  - Sailors × Reserves
- WHERE S.sid=R.sid AND R.bid=103
  - $\sigma_{Sailors.sid = Reserves.sid\ and\ Reserves.bid=103}$
- SELECT S.sname
  FROM Sailors S, Reserves R
  WHERE S.sid=R.sid AND R.bid=103
  - $\pi_{sname}(\sigma_{Sailors.sid = Reserves.sid\ and\ Reserves.bid=103}(\text{Sailors} \times \text{Reserves}))$
- It is often helpful to write an SQL query in the same order (FROM, WHERE, SELECT).

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?

Expressions in SELECT

- Both the select and condition statements can include arithmetical operations
  - Arithmetical operations can be performed on numeric data in the **SELECT** statement
  - Arithmetical operations can also be included in the operands to Boolean operators in the **WHERE** condition
- Column names in the result table can be named, or renamed using the **AS** keyword

Renaming Using AS

- Return the customer id of each invoice, and an estimate of the original cost (tax excluded)
  - Assuming the sales tax rate is 12%

```
SELECT InvoiceId, CustomerId AS cid,
    Total / 1.12 AS original_cost
FROM invoices
```

This query would return a table whose columns were titled *InvoiceId*, *cid*, and *original_cost*

Expressions and Strings

```
SELECT InvoiceId, CustomerId as cid, BillingCity, Total/1.12 as original_cost
FROM invoices
WHERE BillingCity LIKE 'D_%n'
```

- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters. case sensitvity Oracle on Strings

Pattern Matching

- SQL provides pattern matching support with the **LIKE** operator and two symbols
  - The **%** symbol stands for zero or more arbitrary characters
  - The **_** symbol stands for exactly one arbitrary character
- Implementations of **LIKE** vary
  - **'Buffy' LIKE 'Buffy '**

Like This ...

- Return the SINs, and first and last names of customers whose last name is similar to 'Smith'

```
SELECT sin, fname, lname
FROM Customer
WHERE lname LIKE 'Sm_t%'
```

This query would return customers whose last name is *Smit*, or *Smith*, or *Smythe*, or *Smittee*, or *Smut* (!) but not *Smeath*, or *Smart*, or *Smt*

Operations with NULL

- 23 < NULL
  - UNKNOWN
- NULL >= 47
  - UNKNOWN
- NULL = NULL
  - UNKNOWN
- NULL IS NULL
  - TRUE
- 23 + NULL - 3
  - NULL
- NULL * 0
  - NULL

So don't do this to find nulls!

More Fun with Nulls

- Let's say there are 2,753 records in Customer
  - How many rows would a query return that selects customers who are 45 or younger, or older than

```
SELECT sin, fname, lname
FROM Customer
WHERE age <= 45 OR age > 45
```

This should return 2,753 customers, but what happens when we don't know a customer's age (i.e. it is *null*)?

null <= 45 = *unknown* and *null* > 45 = *unknown* and then *unknown or unknown* = *unknown* which is treated as *false*!

Dates and Times

- The SQL standard supports operations on dates
  - Date attributes are created of type `DATE`
- Dates are specified as follows: `'yyyy-mm-dd'`
  - e.g. `'1066-10-14'`
- Dates can be compared
- SQL gives support for `TIME` and `TIMESTAMP` types
  - The `TIMESTAMP` type combines dates and times
- Exact specification, and allowable formats for dates and times varies by SQL implementation

Evaluation of Unknown

- Truth values for *unknown* results
  - *true* **OR** *unknown* = *true*,
  - *false* **OR** *unknown* = *unknown*,
  - *unknown* **OR** *unknown* = *unknown* ,
  - *true* **AND** *unknown* = *unknown*,
  - *false* **AND** *unknown* = *false*,
  - *unknown* **AND** *unknown* = *unknown*
  - **NOT** *unknown* = *unknown*
- The result of a `WHERE` clause is treated as *false* if it evaluates to unknown

Ordering Output

- The output of an SQL query can be ordered
  - By any number of attributes, and
  - In either ascending or descending order
- Return the name and incomes of customers, ordered alphabetically by name

```
SELECT lname, fname, income
FROM Customer
ORDER BY lname, fname
```

The default is to use ascending order, the keywords *ASC* and *DESC*, following the column name, sets the order

Null Values

- A database may contain `NULL` values where
  - Data is unknown, or
  - Data does not exist for an attribute for a particular row
- There are special operators to test for null values
  - **IS NULL** tests for the presence of nulls and
  - **IS NOT NULL** tests for the absence of nulls
- Other comparisons with nulls evaluate to `UNKNOWN`
  - Even when comparing two nulls
  - Except that two rows are evaluated as duplicates, if all their corresponding attributes are equal or both null
- Arithmetic operations on nulls return `NULL`

More Fun with Nulls

- Let's say there are 2,753 records in Customer
  - How many rows would a query return that selects customers who are 45 or younger, or older than

```
SELECT sin, fname, lname
FROM Customer
WHERE age <= 45 OR age > 45
```

This should return 2,753 customers, but what happens when we don't know a customer's age (i.e. it is *null*)?

Set Operations and Joins

Set Operations

- SQL supports union, intersection and set difference operations
  - Called **UNION**, **INTERSECT**, and **EXCEPT**
  - These operations must be performed on *union compatible* tables
- Although these operations are supported in the SQL standard, implementations may vary
  - **EXCEPT** may not be implemented
    - When it is, it is sometimes called **MINUS**

No Account in Robson

- Find the SINs of customers who have an account in the Lonsdale branch
- But who do *not* have an account in the Robson branch

```
SELECT O.osin
FROM Owns O, Account A
WHERE O.accnum = A.accnum AND A.brname =
    'Lonsdale' AND A.brname <> 'Robson'
```

No Account in Robson Again

- Find the SINs of customers who have an account in the Lonsdale branch but don't have one in Robson
- **And get it right this time!**

```
SELECT O1.osin
FROM Owns O1, Account A1,
     Owns O2, Account A2
WHERE O1.osin = O2.osin AND O1.accnum =
  A1.accnum AND O2.accnum = A2.accnum
  AND A1.brname = 'Lonsdale' AND
  A2.brname <> 'Robson'
```

Find sid's of sailors who've reserved a red or a green boat

- If we replace OR by AND in the first version, what do we get?

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

- Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='green'
```

No Account in Robson

- Find the SINs of customers who have an account in the Lonsdale branch
- But who do *not* have an account in the Robson branch

```
SELECT O.osin
FROM Owns O, Account A
WHERE O.accnum = A.accnum AND A.brname =
    'Lonsdale' AND A.brname <> 'Robson'
```

What does this query return?

No Account in Robson Again

- Find the SINs of customers who have an account in the Lonsdale branch but don't have one in Robson
- **And get it right this time!**

```
SELECT O1.osin
FROM Owns O1, Account A1,
     Owns O2, Account A2
WHERE O1.osin = O2.osin AND O1.accnum =
  A1.accnum AND O2.accnum = A2.accnum
  AND A1.brname = 'Lonsdale' AND
  A2.brname <> 'Robson'
```

What does this query return?

Find sid's of sailors who've reserved a red and a green boat

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
     Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
  AND S.sid=R2.sid AND R2.bid=B2.bid
  AND (B1.color='red' AND B2.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.sid AND R.bid=B.bid
        AND B.color='green'
```

No Account in Robson

- Find the SINs of customers who have an account in the Lonsdale branch
- But who do *not* have an account in the Robson branch


```
SELECT O.osin
FROM Owns O, Account A
WHERE O.accnum = A.accnum AND A.brname =
    'Lonsdale' AND A.brname <> 'Robson'
```

What does this query return?

Customers who own an account at Lonsdale (and note that Lonsdale is not the same as Robson …)

No Account in Robson Again

- Find the SINs of customers who have an account in the Lonsdale branch but don't have one in Robson
- **And get it right this time!**


```
SELECT O1.osin
FROM Owns O1, Account A1,
     Owns O2, Account A2
WHERE O1.osin = O2.osin AND O1.accnum =
  A1.accnum AND O2.accnum = A2.accnum
  AND A1.brname = 'Lonsdale' AND
  A2.brname <> 'Robson'
```

What does this query return?

The SINs of customers who own an account at Lonsdale and *any* account that isn't at the Robson branch

Accounts EXCEPT Robson

- *This time* find the SINs of customers who have an account at the Lonsdale branch but not at Robson

Accounts EXCEPT Robson

- *This time* find the SINs of customers who have an account at the Lonsdale branch but not at Robson

```
SELECT O1.osin
FROM Owns O1, Account A1
WHERE A1.accnum = O1.accnum AND
    A1.brname = 'Lonsdale'
EXCEPT
SELECT O2.osin
FROM Owns O2, Account A2
WHERE A2.accnum = O2.accnum AND
    A2.brname = 'Robson'
```

Set Operations and Duplicates

- Unlike other SQL operations, **UNION**, **INTERSECT**, and **EXCEPT** queries eliminate duplicates by default
- SQL allows duplicates to be *retained* in these three operations using the **ALL** keyword
- If **ALL** is used, and there are *m* copies of a row in the upper query and *n* copies in the lower
  - **UNION** returns *m* + *n* copies
  - **INTERSECT** returns min(*m*, *n*) copies?
  - **EXCEPT** returns *m* – *n* copies?
- It is generally advisable not to specify **ALL**

EXCEPT, without ALL

$\pi_{fname,lname}$(Customer)

fname	lname
Arnold	Alliteration
Bob	Boyd
Bob	Boyd
Charlie	Clements
Bob	Boyd

$\pi_{fname,lname}$(Employee)

fname	lname
Bob	Boyd
Charlie	Clements
Susie	SummerTree
Desmond	Dorchester

```
SELECT fname, lname
FROM Customer
EXCEPT
SELECT fname, lname
FROM Employee
```

fname	lname
Arnold	Alliteration

EXCEPT ALL

$\pi_{fname,lname}$(Customer)

fname	lname
Arnold	Alliteration
Bob	Boyd
Bob	Boyd
Charlie	Clements
Bob	Boyd

$\pi_{fname,lname}$(Employee)

fname	lname
Bob	Boyd
Charlie	Clements
Susie	SummerTree
Desmond	Dorchester

```
SELECT fname, lname
FROM Customer
EXCEPT ALL
SELECT fname, lname
FROM Employee
```

fname	lname
Arnold	Alliteration
Bob	Boyd
Bob	Boyd

Exercise 5.2

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. Find the pnames of parts for which there is some supplier.
2. Find the sids of suppliers who supply a red part or a green part.
3. Find the sids of suppliers who supply a red part and a green part.

```
1.
select DISTINCT P.pname
from Parts P, Suppliers S, Catalog C
where C.pid=P.pid and C.sid=S.sid;
2.
select DISTINCT C.sid
from Parts P, Catalog C
where P.pid=C.pid and (P.color='red' or P.color='green');
3.
select DISTINCT C.sid
from Parts P, Catalog C
where P.pid=C.pid and P.color='red'
INTERSECT
select DISTINCT C.sid
from Parts P, Catalog C
where P.pid=C.pid and P.color='green';
```

Join

- Expressed in FROM clause and WHERE clause.
- Forms the subset of the *Cartesian product* of all relations listed in the FROM clause that satisfies the WHERE condition:

```
SELECT *
FROM   Sailors, Reserves
WHERE Sailors.sid = Reserves.sid;
```

Join in SQL

- Since joins are so common, SQL provides JOIN as a shorthand.

```
SELECT *
    FROM Sailors JOIN Reserves ON
        Sailors.sid = Reserves.sid;
```

- NATURAL JOIN produces the natural join of the two input tables, i.e. an equi-join on all attributes common to the input tables.

```
SELECT *
FROM Sailors NATURAL  JOIN Reserves;
```

Join Types

- **INNER** – only includes records where attributes from both tables meet the join condition
- **LEFT OUTER** – includes records from the *left table* that do not meet the join condition
- **RIGHT OUTER** – includes records from the *right table* that do not meet the join condition
- **FULL OUTER** – includes records from *both tables* that do not meet the join condition
- In outer joins, tables are padded with **NULL** values for the missing attributes where necessary

Left Outer Join Results

empSIN	salary	income
111	29000	29000
222	73000	null
333	48000	null
444	83000	150000
555	48000	null
666	53000	53000
777	3200	null

all the employees

incomes of employees who are customers

```
SELECT E.empSIN, E.salary, C.income
FROM Employee E LEFT OUTER JOIN
   Customer C ON E.empSIN = C.osin
```

Nested Queries

- A nested query is a query that contains an embedded query, called a *subquery*
- Subqueries can appear in a number of places
  - In the **FROM** clause,
  - In the **WHERE** clause, and
  - In the **HAVING** clause
- Subqueries referred to in the **WHERE** clause are often used in additional set operations
- Multiple levels of query nesting are allowed

Join Conditions

- **NATURAL** – equality on all attributes in common
  - Similar to a relational algebra natural join
- **USING** (A_1, …, A_n) – equality on all the attributes in the attribute list
  - Similar to a relational algebra theta join on equality
- **ON**(condition) – join using condition
  - Similar to a relational algebra theta join
- Join conditions can be applied to outer or inner joins
  - If no condition is specified for an inner join the Cartesian product is returned
  - A condition must be specified for an outer join

Customer Accounts

- Return the SINs, first and last names, and account numbers of customers who own accounts

```
SELECT C.osin, c.fname, C.lname, O.accnum
FROM Customer C NATURAL INNER JOIN Owns O
```

No records will be returned for customers who do not have accounts

A natural join can be used here because the Owns and Customer table both contain attributes called *osin*

Additional Set Operations

- IN
- NOT IN
- EXISTS
- NOT EXISTS
- UNIQUE
- ANY
- ALL

Employees who are Customers

- Return the SINs and salaries of all employees, if they are customers also return their income

```
SELECT E.empSIN, E.salary, C.income
FROM Employee E LEFT OUTER JOIN
   Customer C ON E.empSIN = C.osin
```

- A *left outer join* is often preferred to a *right outer join*
  - As any nulls will appear on the right-hand side of the result

In this example the income column will contain nulls for those employees who are not also customers

Nested SQL Queries

Accounts IN Lonsdale

- Find the SINs, ages and incomes, of customers who have an account at the Lonsdale branch

```
SELECT C.osin, C.age, C.income
FROM Customer C
WHERE C.osin IN
    (SELECT O.osin
     FROM Account A, Owns O
     WHERE A.accnum = O.accnum AND
  A.brname = 'Lonsdale')
```

Replacing IN with NOT IN in this query would return the customers who do not have an account at Lonsdale

Uncorrelated Queries

- The query shown previously contains an *uncorrelated*, or *independent*, subquery
  - The subquery does not contain references to attributes of the outer query
- An independent subquery can be evaluated before evaluation of the outer query
  - And needs to be evaluated only once
    - The subquery result can be checked for each row of the outer query
  - The cost is the cost for performing the subquery (once) and the cost of scanning the outer relation

EXISTing Lonsdale Accounts

- Find the SINs ages and income, of customers who have an account at the Lonsdale branch

```
SELECT C.osin, C.age, C.income
FROM Customer C
WHERE EXISTS
    (SELECT *
     FROM Account A, Owns O
     WHERE C.osin = O.osin AND
      A.accnum = O.accnum AND
  A.brname = 'Lonsdale')
```

EXISTS and NOT EXISTS test to see whether the associated subquery is non-empty or empty

Correlated Queries

- The previous query contains a *correlated* subquery
  - The subquery contains references to attributes of the outer query
    - … `WHERE C.osin = O.osin` …
  - The subquery therefore has to be evaluated once *for each row* in the outer query
    - i.e. for each row in the Customer table
- Correlated queries are often inefficient
  - Unfortunately some DBMSs do not distinguish between the evaluation of correlated and uncorrelated queries

Division using NOT EXISTS

- Find the names and SINs of customers who have an account in all branches
  - This is an example of a query that would use division
  - However division is often not implemented in SQL
  - But can be computed using NOT EXISTS or EXCEPT
- To build a division query start by finding all the branch names
  - As this part is easy!

```
SELECT B.brname
FROM Branch B
```

More Division and NOT EXISTS

- We can also find all the branches that a particular customer has an account in

```
SELECT A.brname
FROM Account A, Owns O
WHERE O.osin = some_customer AND
    O.accNum = A.accnum
```

OK, so I'm cheating here by putting in "some customer" but we'll fix that part later using a correlated subquery

Magic with EXCEPT

SQ1 – A list of all branch names

EXCEPT

SQ2 – A list of branch names that a customer has an account at

The result contains all the branches that a customer does *not* have an account at, if the customer has an account at every branch then this result is empty.

And Finally …

- Putting it all together we have

And Finally …

- Putting it all together we have

```
SELECT C.osin , C.fname, C.lname
FROM Customer C
WHERE NOT EXISTS
    (SELECT B.brname
      FROM Branch B
     EXCEPT
     SELECT A.brname
      FROM Account A, Owns O
      WHERE O.osin = C.osin AND
          O.accnum = A.accnum);
```

Exercise 5.2 ctd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL. You can use NOT EXISTS.

1. Find the sids of suppliers who supply only red parts.
2. Find the snames of suppliers who supply every part. (difficult)

```
1.
select DISTINCT C.sid from Parts P, Catalog C
where C.pid=P.pid and P.color='red' and NOT EXISTS
(select * from Parts P1, Catalog C1 where C1.sid=C.sid and
   C1.pid=P1.pid and P1.color<>'red');

select DISTINCT C.sid from Parts P, Catalog C
where C.pid=P.pid and P.color='red'
EXCEPT
select DISTINCT C.sid from Parts P, Catalog C
where C.pid=P.pid and P.color<>'red';
2.
select DISTINCT S.sname from Suppliers S
where NOT EXISTS
   (select DISTINCT P.pid from Parts P
    EXCEPT
    select DISTINCT P1.pid from Parts P1, Catalog C where
   C.sid=S.sid and C.pid=P1.pid
   );
```

ANY and ALL

- The **ANY** (**SOME** in some DBMS) and **ALL** keywords allow comparisons to be made to *sets* of values
  - The keywords must be preceded by a Boolean operator
    - <, <=, =, <>, >=, or >
  - **ANY** and **ALL** are used in the **WHERE** clause to make a comparison to a subquery
- **ANY** and **ALL** can be compared to **IN** and **NOT IN**
  - **IN** is equivalent to **= ANY**
  - **NOT IN** is equivalent to **<> ALL**

Aggregations

UNIQUE and NOT UNIQUE

- The **UNIQUE** operator tests to see if there are no duplicate records in a query
  - **UNIQUE** returns **TRUE** if no row appears twice in the answer to the subquery
    - Two rows are equal if, for each attribute, the attribute values in one row equal the values in the second
  - Also, if the subquery contains only one row *or is empty* **UNIQUE** returns **TRUE**
- **NOT UNIQUE** tests to see if there are at least two identical rows in the subquery

ANYone Richer Than Bruce

- Find the SINs and names, of customers who earn more than *any* customer called Bruce

```
SELECT C1.osin, C1.fname, C1.lname
FROM Customer C1
WHERE C1.income > ANY
   (SELECT Bru.income
    FROM Customer Bru
      WHERE Bru.fname = 'Bruce')
```

Customers in the result table must have incomes greater than at least one of the rows in the subquery result

Aggregate Operators

- SQL has a number of operators which compute *aggregate* values of columns
  - **COUNT** – the number of values in a column
  - **SUM** – the sum of the values in a column
  - **AVG** – the average of the values in a column
  - **MAX** – the maximum value in a column
  - **MIN** – the minimum value in a column
- **DISTINCT** can be applied to any of these operations but this should only be done with care!

UNIQUE Accounts

- Find the SINs, first names, and last names of customers who have only one account

```
SELECT C.osin, C.fname, C.lname
FROM Customer C
WHERE UNIQUE
   (SELECT O.osin
    FROM Owns O
      WHERE C.osin = O.osin)
```

This is a correlated query, using NOT UNIQUE would return customers with at least two accounts

What happens if a customer doesn't have an account?

Richer Than ALL Bruce(s)

- Find the SINs and names, of customers who earn more than *all* customers called Bruce

```
SELECT C1.osin, C1.fname, C1.lname
FROM Customer C1
WHERE C1.income > ALL
   (SELECT Bru.income
    FROM Customer Bru
      WHERE Bru.fname = 'Bruce')
```

If there were no customers called Bruce this query would return all customers

AVG Income

- Find the average customer income

```
SELECT AVG (income) AS average_income
FROM Customer
```

The average column will be nameless unless it is given a name, hence the AS statement

Note that this is a query where using DISTINCT would presumably not give the intended results, as multiple identical incomes would only be included once

How Many Names For Smiths?

- Find the number of *different* first names for customers whose last name is Smith

```
SELECT COUNT (DISTINCT fname) AS sm_names
FROM Customer
WHERE lname = 'Smith' OR lname = 'smith'
```

In this query it is important to use DISTINCT, otherwise the query will simply count the number of people whose last name is Smith

Exercise 5.2 ctd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following query in SQL:

1. Find the average cost of Part 70 (over all suppliers of Part 70).
2. Find the names of suppliers who charge more for Part 70 than the average cost of Part 70.
3. Find the sids of suppliers who charge more for some part than the average cost of that part.

Counting Accounts By Branch

- Find the number of accounts held by each branch

```
SELECT brName, COUNT (accnum) AS num_acc
FROM Account
GROUP BY brname
```

Every column that appears in the SELECT list that is not an aggregation must also appear in the group list

Aggregations and SELECT

- Find the SIN and income of the customer with the lowest income

```
SELECT osin, MIN (income)
FROM Customer
```

wrong

What is wrong with this query?

There may be two people with the same minimum income

This query is therefore illegal, if any aggregation is used in the SELECT clause it can only contain aggregations, unless the query also contains a GROUP BY clause

```
1.
select Avg(C.cost) from Catalog C
where C.pid=70;
2.
select S.sname from Suppliers S, Catalog C
where C.sid=S.sid and C.pid=70 and C.cost >
  (select Avg(C1.cost) from Catalog C1
  where C1.pid=70);
3.
select S.sid, C.pid from Suppliers S, Catalog C
where C.sid=S.sid and C.cost >
  (select Avg(C1.cost)from Catalog C1
  where C1.pid=C.pid);
```

Query Example – HAVING

- Find the number of accounts held by each branch
  - Only for branches that have budgets over $500,000 and total account balances greater than $1,000,000

```
SELECT B.brname, COUNT (A.accnum) AS accs
FROM Account A, Branch B
WHERE A.brname = B.brname AND
    B.budget > 500000
GROUP BY B.brname
HAVING SUM (A.balance) > 1000000
```

The HAVING clause is essentially a condition that is applied to each group rather than to each row

Aggregations and SELECT

- In the previous query a single aggregate value was, potentially, matched to a *set* of values
- In the query below, a set of pairs is returned
  - The income in each pair matches the single value returned by the subquery

```
SELECT C1.osin, C1.income
FROM Customer C1
WHERE C1.income =
   (SELECT MIN (C2.income)
    FROM Customer C2)
```

Grouping Aggregations

- Consider a query to find out how many accounts there are in *each* branch
  - This requires that there are multiple counts, one for each branch
  - In other words a series of aggregations, based on the value of the branch name
  - Given the syntax shown so far this is not possible to achieve in one query
  - But it is possible using the **GROUP BY** clause

Order of Evaluation (Reprise)

- Create the Cartesian product of the tables in the **FROM** clause
- Remove rows not meeting the **WHERE** condition
- Remove columns not in the **SELECT** clause
- Sort records into groups by the **GROUP BY** clause
- Remove groups not meeting the **HAVING** clause
- Create one row for each group
- Eliminate duplicates if **DISTINCT** is specified

Exercise 5.2 cont'd.

Consider the following schema.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

The Catalog lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. For every supplier that supplies only green parts, print the name of the supplier and the total number of parts that she supplies.
2. For every supplier that supplies at least a green part and at least a red part, print the name(s) and price of the most expensive part(s) that she supplies.

```
select S.sname, COUNT(C.pid)
from Suppliers S, Catalog C
where C.sid=S.sid and S.sid IN
   (
   select DISTINCT C1.sid
   from Parts P1, Catalog C1
   where C1.pid=P1.pid and P1.color='red'
   EXCEPT
   select DISTINCT C2.sid
   from Parts P2, Catalog C2
   where C2.pid=P2.pid and P2.color<>'red'
   )
group by S.sid;
```

```
select S.sname, P.pname, C.cost
from Suppliers S, Parts P, Catalog C
where C.sid=S.sid and P.pid=C.pid and S.sid in
   (
    select DISTINCT C1.sid
    from Parts P1, Catalog C1
    where P1.pid=C1.pid and P1.color='red'
    INTERSECT
    select DISTINCT C2.sid
    from Parts P2, Catalog C2
    where P2.pid=C2.pid and P2.color='green'
   )
   and C.cost=(
    select max(C3.cost)
    from Catalog C3
    where C3.sid=S.sid
   );
```

Readings

- **RG Chapter 5**
- GUW Chapter 6.1-6.4

SDSC 5003
SQL Constraints & Triggers

Instructor: Yu Yang

yuyang@cityu.edu.hk

SQL Constraints

- Constraints
  - Primary Key
  - Foreign Key
  - General table constraints
  - Domain constraints
  - Assertions
- Triggers

Primary and Foreign Keys

Primary Key Constraints

- Every table should have a primary key
- When a primary key constraint is created it specifies that:
  - The attributes of the primary key cannot be null
  - The primary key must be unique
- Violating a primary key causes the violating update to be rejected

SQL Server Primary Keys

- A primary key constraint can be specified in a number of ways
  - Create or alter table statement
    - PRIMARY KEY (<attribute>)
  - Selecting the attribute(s) and choosing the primary key menu item in the context menu
- By default SQL server creates a clustered index on the primary key attributes
  - Clustered indexes support range queries

Foreign Key Constraints

- Represents a relationship between two tables
- If table *R* contains a *foreign key*, on attributes {*a*}, that references table *S*
  - {*a*} must correspond to the primary key of *S*
    - Must have the same number of attributes, and
    - The same domains
  - Values for {*a*} in *R* must also exist in *S* except that
    - If {*a*} is not part of the primary key of *R* it may be *null*
  - There may be values for {*a*} in *S* that are not in *R*

Foreign Key Specification

- Foreign keys specify the actions to be taken if referenced records are updated or deleted
  - For example, create a foreign key in Account that references Branch
    - Assign accounts of a deleted branch to Granville
    - Cascade any change in branch names

```
…
brName CHAR(20) DEFAULT 'Granville',
FOREIGN KEY (brName) REFERENCES Branch
   ON DELETE SET DEFAULT
   ON UPDATE CASCADE)
```

Cascading Changes

- It is possible that there can be a chain of foreign key dependencies
  - e.g. branches, accounts, transactions
- A cascading deletion in one table may cause similar deletions in a table that references it
  - If any cascading deletion or update causes a violation, *the entire transaction* is aborted

Foreign Keys in DBMS

- Foreign keys can often be specified in a number of different ways
  - A separate foreign key statement
    - Very similar to the SQL standard
  - A References statement following a column name
  - Using the GUI
- Updates and deletions in the referenced table can be set to a number of actions
  - No action (the default) – the transaction is rejected
  - Cascade, set null or set default

Referencing non Primary Keys

- By default SQL foreign keys reference the primary key (of the referenced table)
- It is possible to reference a list of attributes
  - The list must be specified after the name of the referenced table
  - The specified list of attributes must be declared as a candidate key of the referenced table (UNIQUE)
    - In SQL Server in Create or Alter Table statement or
    - Select Indexes/Keys from the context menu

General Constraints

General Constraints

- A general or *table* constraint is a constraint over a single table
  - Included in a table's `CREATE TABLE` statement
  - Table constraints may refer to other tables
- Defined with the `CHECK` keyword followed by a description of the constraint
  - The constraint description is a Boolean expression, evaluating to true or false
  - If the condition evaluates to false the update is rejected

Constraint Example

- Check that a customer's age is greater than 18, and that a customer is not an employee

```
CREATE TABLE Customer
  (sin  CHAR(11),
   …,
   income REAL,
   PRIMARY KEY (sin),
   CONSTRAINT CustAge CHECK (age > 18),
   CONSTRAINT notEmp CHECK (sin NOT IN
   (SELECT empSIN FROM Employee)))
```

this wouldn't work

Check Constraints in DBMS

- Check constraint are often limited to comparisons with *scalar expressions*
  - That is, expressions that contain single values
  - Expressions that contain SQL queries are therefore not allowed
- Check expressions can include comparisons to results of User Defined Functions (UDFs) **for some DBMS**
  - As long as the function returns a scalar value

UDF (in SQL Server)

- A UDF is an SQL Server function
- UDFs purpose and output vary widely
  - They can return scalar values or tables
  - They can be written in T-SQL or a .NET language
- UDF are useful for a number of reasons
  - Modular programming
  - Faster execution
  - Reduced network traffic

UDFs and Check Constraints

```
CREATE FUNCTION checkEmpNotCustomer()
RETURNS int
AS
BEGIN
      DECLARE @result int
      if (select COUNT(*) from Employee e, Customer c
          where c.cid = e.sin) = 0
            SET @result = 1
      ELSE
            SET @result = 0
      RETURN @result
END
```

The corresponding check expression is
dbo.checkEmpNotCustomer = 1

Domains and Types

Domain Constraints

- New domains can be created using the **CREATE DOMAIN** statement
  - Each such domain must have an underlying source type (i.e. an SQL base type)
  - A domain must have a name, base type, a restriction, and a default optional value
    - The restriction is defined with a **CHECK** statement
- Domains are part of the DB schema but are not attached to individual table schemata

Domain Constraint Example

- Create a domain for minors, who have ages between 0 and 18
  - Make the default age 10 (!)

```
CREATE DOMAIN minorAge INTEGER DEFAULT 10
  CHECK (VALUE > 0 AND VALUE <= 18)
```

Using Domain Constraints

- A domain can be used instead of one of the base types in a **CREATE TABLE** statement
  - Comparisons between two domains are made in terms of the underlying base types
    - e.g. comparing an age with an account number domain simply compares two integers
- SQL also allows distinct types to be created
  - Types are distinct so that values of different types cannot be compared

Creating Types

- The SQL **CREATE TYPE** clause defines new types
  - To create distinct age and account number types:
    - `CREATE TYPE Ages AS INTEGER`
    - `CREATE TYPE Accounts AS INTEGER`
  - Assignments, or comparisons between ages and account numbers would now be illegal
    - Although it is possible to *cast* one type to another

SQL Server Base Types

bigint	binary(n)	bit	char(n)
8 bytes	fixed length binary	single bit	fixed length
date	datetime	datetime2	datetimeoffset
yyyy-mm-dd	date and time	larger range	includes time zones
decimal	float	image	int
select precision, scale	approximate	variable length binary	4 bytes
money	nchar(n)	ntext	numeric
8 byte monetary	fixed length unicode	variable length unicode	equivalent to decimal
nvarchar(n | max)	real	smalldatetime	smallint
variable length unicode	4 byte float	limited range	2 bytes
smallmoney	sql_variant	text	time
4 byte monetary	allows many types	variable length	time (not date)
tinyint	uniqueidentifier	varbinary(n | max)	varchar(n | max)
1 byte	used for keys	variable length binary	variable length

Creating SQL Server Types

- SQL Server allows types to be created using syntax similar to the SQL standard
  - `CREATE TYPE SSN FROM varchar(11);`

Assertions

Assertion Limitations

- There are some constraints that cannot be modeled with table constraints or assertions
  - What if there were participation constraints between customers and accounts?
    - Every customer must have at least one account and every account must be held by at least one customer
  - An assertion *could* be created to check this situation
    - But would prevent new customers or accounts being added!

Why Use Triggers?

- Triggers can implement business rules
  - e.g. creating a new loan when a customer's account is overdrawn
- Triggers may also be used to maintain data in related database tables
  - e.g. Updating derived attributes when underlying data is changed, or maintaining summary data
- Many trigger actions can also be performed by *stored procedures (not supported by SQLite)*

Assertions

- Table constraints apply to only one table
- Assertions are constraints that are separate from `CREATE TABLE` statements
  - Similar to domain constraints, they are separate statements in the DB schema
  - Assertions are tested *whenever the DB is updated*
    - Therefore they may introduce significant overhead
- T-SQL (Microsoft SQL) does not implement assertions

Triggers

Trigger Components

- Event
  - A specified modification to the DB
    - May be an insert, deletion, or change
    - May be limited to specific tables
    - A trigger may *fire* before or after the transaction
- Condition
- Action

Example Assertion

- Check that a branch's assets are greater than the total account balances held in the branch

```
CREATE ASSERTION assetCoverage
CHECK (NOT EXISTS
      (SELECT *
       FROM Branch B
       WHERE B.assets <
         (SELECT SUM (A.balance)
          FROM Account A
          WHERE A.brName = B.brName)))
```

This prevents changes to both the Branch and Account tables

Triggers

- A trigger is a procedure that is invoked by the DBMS as a response to a specified change
  - A DB that has a set of associated triggers is referred to as an *active database*
  - Triggers are available in most current commercial DB products
    - And are part of the SQL standard
- Triggers carry out *actions* when their triggering conditions are met
  - Generally SQL constraints only reject transactions

Trigger Components

- Event
- Condition
  - A Boolean expression or a query
    - If the query answer set is non-empty it evaluates to true, otherwise false
    - If the condition is true the trigger action occurs
- Action

Trigger Components

- Event
- Condition
- Action
  - A trigger's action can be very far-ranging, e.g.
    - Execute queries
    - Make modifications to the DB
    - Create new tables
    - Call host-language procedures

Trigger Syntax

- The SQL standard gives a syntax for triggers
  - In practice, trigger syntax varies from system to system
  - Many features of triggers are common to the major DBMS products, and the SQL standard

SQLite Trigger Example

```
CREATE TRIGGER reminder
      AFTER INSERT ON Customer
BEGIN
      SELECT RAISE(ABORT, 'Notify Customer Relations');
END;
```

- This trigger sends a message
  - When customer data is added
- The trigger has three components
  - Event – insertion into the Customer table
  - Condition – always!
  - Action – stop insertion and print a message
    - Performed for every row affected in Event (**time-consuming!**)

SQLite Trigger Syntax

table that the trigger applies to

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
   [BEFORE | AFTER | INSTEAD OF] [INSERT | UPDATE | DELETE]
   ON table_name
   [WHEN condition]
BEGIN
statements;
END;
```

SQLite Trigger Syntax

table that the trigger applies to

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
   [BEFORE | AFTER | INSTEAD OF] [INSERT | UPDATE | DELETE]
   ON table_name
   [WHEN condition]
BEGIN
statements;
END;
```

event

SQLite Trigger Syntax

table that the trigger applies to

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
   [BEFORE | AFTER | INSTEAD OF] [INSERT | UPDATE | DELETE]
   ON table_name
   [WHEN condition]
BEGIN
statements;
END;
```

event

Actions performed

SQLite Trigger Events

- Triggers can apply changes to tables
  - Deletion, insertion or update
  - Or any combination of the three
- Triggers can be specified as
  - BEFORE/AFTER
    - Occur before/after the transaction has taken place
  - INSTEAD OF
    - The trigger takes the place of the triggering statement
    - Only one instead of trigger is allowed for each insert, update or delete statement for a table

Trigger Actions

- A trigger's action can be almost anything (for some DBMS)
  - SQL statements
  - Here is an example of the trigger that sends an email (in SQL Server)

```
CREATE TRIGGER reminder2
ON Customer
AFTER INSERT
AS
EXEC msdb.dbo.sp_send_dbmail
      @profile_name = 'AdventureWorks Administrator',
      @recipients = 'danw@Adventure-Works.com',
      @body = 'Don"t forget to print a report for the sales force.',
      @subject = 'Reminder';
```

Trigger Conditions

- Trigger conditions are contained in an if statement
  - IF should be followed by a Boolean expression
  - The Boolean expression is commonly an SQL expression
    - e.g. IF EXISTS(subquery)
- If statements in triggers may have an else clause

Balance Limit

- Reject the insertion of new accounts whose balance is greater than twice the current highest balance

```
CREATE TRIGGER balance
    BEFORE INSERT on Transactions
    WHEN NOT EXISTS (select * from Account where balance*2 <
    NEW.balance)
BEGIN
    SELECT RAISE(ABORT, 'This transaction is invalid. Insufficient
    balance.');
END;
```

Could also be implemented as a check constraint with a UDF (for some DBMS)

Invalid Transactions

- Forbid invalid transactions where the amount is greater than the balance of the payer

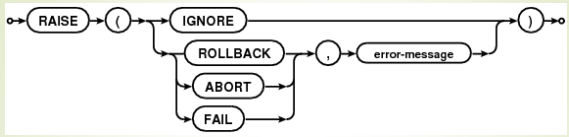
```
CREATE TRIGGER balance1
    BEFORE INSERT on Transactions
    WHEN (select balance from Account where aid=NEW.aidFrom) <
    NEW.amount
BEGIN
    SELECT RAISE(ABORT, 'This transaction is invalid. Insufficient
    balance.');
END;
```

INSERT INTO AccountInfo

```
create view AccountInfo (aid, cid, name, brName, balance)
as
select A.aid, A.cid, C.name, A.brName, A.balance
from Account A, Customer C
where A.cid=C.cid;
```

```
CREATE TRIGGER insert_account_info1
   INSTEAD OF INSERT ON AccountInfo
      WHEN NOT EXISTS (select * from Branch where brName=NEW.brN
ame)
BEGIN
   INSERT INTO Branch(brName)
      VALUES (NEW.brName);
END;
```

SQLite RAISE Function



- RAISE(ROLLBACK,...), RAISE(ABORT,...) or RAISE(FAIL,...)
  - Current query terminates
  - The specified ON CONFLICT processing is performed
  - Error message
- RAISE(IGNORE), terminate
  - Remainder of the current trigger
  - Statement that caused the trigger
  - Any subsequent trigger

Balancing Balance

- Add/Subtract the amount of a transaction to the balance of the associated account

```
CREATE TRIGGER balance2
    BEFORE INSERT on Transactions
    WHEN (select balance from Account where aid=NEW.aidFrom) >=
    NEW.amount
BEGIN
    update Account
    set balance=balance-NEW.amount
    where aid=NEW.aidFrom;
    update Account
    set balance=balance+NEW.amount
    where aid=NEW.aidTo;
END;
```

Trigger Summary

- Triggers can be used for many purposes
  - Enforcing business rows
  - Adding functionality to the database
  - ...
- Triggers do carry overhead (FOR EACH ROW in SQLite)
  - Constraints should be enforced by PKs, FKs and simple check constraints where possible
  - Triggers should be made as efficient as possible

NEW and OLD

- NEW
  - Reference to the new row
  - Available in INSERT and UPDATE
- OLD
  - Reference to the old row
  - Available in UPDATE and DELETE

INSTEAD OF

- Work on Views
  - Views in SQLite are read only
  - INSERT, UPDATE and DELETE are not allowed for views
- INSTEAD OF makes views modifiable

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
   INSTEAD OF [DELETE | INSERT | UPDATE OF column_name]
   ON table_name
BEGIN
   statements
END;
```

Readings

- **RG Chapter 5**
- GUW Chapter 7

SDSC 5003

Design Theory

Instructor: Yu Yang
yuyang@cityu.edu.hk

A Toy Example

- **Design theory** is about how to represent your data to avoid anomalies.

Design 1

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Design 2

Student	Course
Mike	5003
Mary	5003
Sam	5003
..	..

Course	Room
5003	LT-5
8005	7303

Four Types of Anomalies - 1

- What's wrong?

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

If every course is in only one room, contains ***redundant*** information!

Four Types of Anomalies - 2

- What's wrong?

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-10
Sam	5003	LT-5
..	..	..

If we update the room number for one tuple, we get inconsistent data = an ***update anomaly***

Four Types of Anomalies - 3

- What's wrong?

Student	Course	Room
..	..	..

If everyone drops the class, we lose what room the class is in! = a ***delete anomaly***

Four Types of Anomalies - 4

- What's wrong?

…	8005	7303

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Similarly, we can't reserve a room without students = an ***insert anomaly***

Elimination of Anomalies

- Is it better?

Student	Course
Mike	5003
Mary	5003
Sam	5003
..	..

Course	Room
5003	LT-5
8005	7303

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

Why this design may be better?
How to find this *decomposition?*

Normal Forms

- 1st Normal Form (1NF) = All tables are flat
- *2nd Normal Form* = 1NF & No non-prime attribute FD part candidate key
- Boyce-Codd Normal Form (BCNF) = no bad FDs
- 3rd , 4th , and 5th Normal Forms = see textbooks

FD (Functional Dependency)

1st Normal Form (1NF)

Student	Courses
Mary	{CS145,CS229}
Joe	{CS145,CS106}
…	…

Violates 1NF.

Student	Courses
Mary	CS145
Mary	CS229
Joe	CS145
Joe	CS106

In 1st NF

1NF Constraint: Types must be atomic!

Functional Dependency

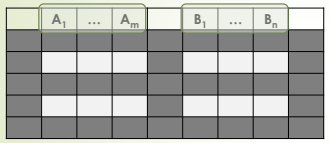
Def: Let A,B be *sets* of attributes
We write A → B or say A ***functionally determines*** B if, for any tuples t_1 and t_2:

$t_1[A] = t_2[A]$ implies $t_1[B] = t_2[B]$

and we call A → B a **functional dependency**

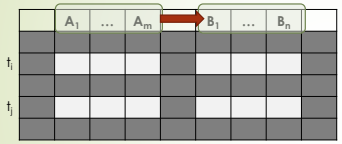
A->B means that
"whenever two tuples agree on A then they agree on B."

A Picture Of FDs



Def (again):
Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R**,

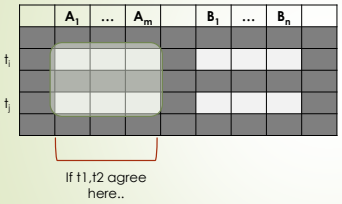
A Picture Of FDs



Defn (again):
Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R**,

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

A Picture Of FDs

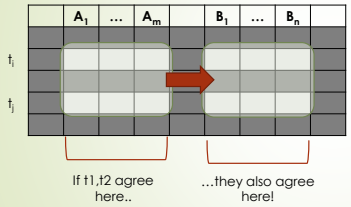


Defn (again):
Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R**,

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

$t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

A Picture Of FDs



Def (again):
Given attribute sets **A={A_1,…,A_m}** and **B = {B_1,…B_n}** in **R**,

The ***functional dependency*** **A→ B on R** holds if for ***any*** t_i,t_j in R:

if $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2]=t_j[A_2]$ AND … AND $t_i[A_m] = t_j[A_m]$

then $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2]=t_j[B_2]$ AND … AND $t_i[B_n] = t_j[B_n]$

Example

An FD holds, or does not hold on a table:

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

Position → Phone

Phone → Position

Phone, Name → Position

Exercise - 1

An FD holds, or does not hold on a table:

Name	Category	Color	Department	Price
Gizmo	Gadget	Green	Toys	49
Tweaker	Gadget	Green	Toys	49
Gizmo	Stationary	Green	Office-supply	59

Name → Color Name → Price

Category → Department Name, Category → Price

Color, Category → Color

Exercise - 2

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

Find at least *three* FDs which **do not hold** on this table:

{ } → { }
{ } → { }
{ } → { }

An Interesting Observation: FD Inference

Provided FDs:

1. Name → Color
2. Category → Department
3. Color, Category → Price

Does it always hold? **Name, Category → Price**

If we find out from application domain that a relation satisfies some FDs, it doesn't mean that we found all the FDs that it satisfies! There could be more FDs implied by the ones we have

Inference Problem

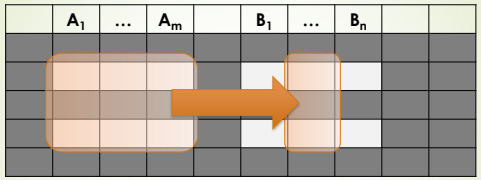
Whether or not a set of FDs imply another FD?

This is called **Inference problem**

Answer: Three simple rules called **Armstrong's Rules.**

1. **Split/Combine,**
2. **Reduction, and**
3. **Transitivity**

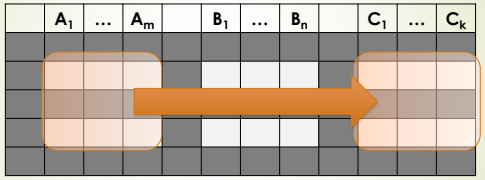
1. Split/Combine



And vice-versa, $A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n
… is equivalent to …

$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

3. Transitive

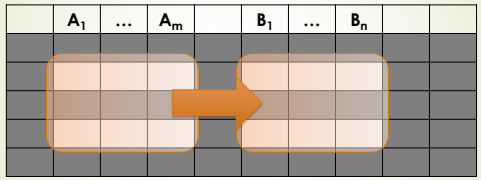


$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and $B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

implies

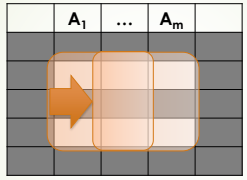
$A_1,\ldots,A_m \rightarrow C_1,\ldots,C_k$

1. Split/Combine



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

2. Reduction/Trivial



$A_1,\ldots,A_m \rightarrow A_j$ for any j=1,…,m

Inferred FDs

Example:

Inferred FDs:

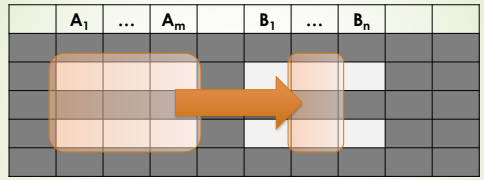
Inferred FD	Rule used
4. Name, Category → Name	?
5. Name, Category → Color	?
6. Name, Category → Category	?
7. Name, Category → Color, Category	?
8. Name, Category → Price	?

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Which / how many other FDs hold?

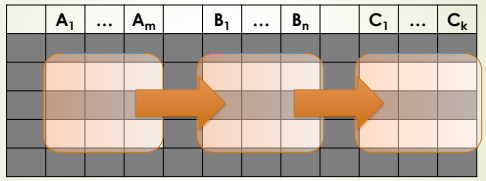
1. Split/Combine



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$

… is equivalent to the following *n* FDs…
$A_1,\ldots,A_m \rightarrow B_i$ for i=1,…,n

3. Transitive



$A_1, \ldots, A_m \rightarrow B_1,\ldots,B_n$ and $B_1,\ldots,B_n \rightarrow C_1,\ldots,C_k$

Inferred FDs

Example:

Inferred FDs:

Inferred FD	Rule used
4. Name, Category → Name	Trivial
5. Name, Category → Color	Transitive (4 -> 1)
6. Name, Category → Category	Trivial
7. Name, Category → Color, Category	Split/combine (5 + 6)
8. Name, Category → Price	Transitive (7 -> 3)

Provided FDs:

1. {Name} → {Color}
2. {Category} → {Dept.}
3. {Color, Category} → {Price}

Can we find an algorithmic way to do this?

Closure of a set of Attributes

Given a set of attributes $A_1, \ldots, A_n$ and a set of FDs **F:**
Then the **closure**, $\{A_1, \ldots, A_n\}^+$ is the set of attributes **B** s.t. $\{A_1, \ldots, A_n\} \rightarrow B$

Example: F =
name → color
category → department
color, category → price

Closures:

$\{name\}^+ = \{name, color\}$
$\{name, category\}^+ = \{name, category, color, dept, price\}$
$\{color\}^+ = \{color\}$

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$ and set of FDs F.
Repeat until X doesn't change; **do**:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F
    and $\{B_1, \ldots, B_n\} \subseteq X$
    then add C to X.
Return X as X^+

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change;
do:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F
    and $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+ =$
$\{name, category\}$

F =
name → color
category → dept
color, category → price

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change;
do:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F
    and $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+ =$
$\{name, category\}$

$\{name, category\}^+ =$
$\{name, category, color\}$

F =
name → color
category → dept
color, category → price

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change;
do:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F
    and $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+ =$
$\{name, category\}$

$\{name, category\}^+ =$
$\{name, category, color\}$

$\{name, category\}^+ =$
$\{name, category, color, dept\}$

F =
name → color
category → dept
color, category → price

Closure Algorithm

Start with $X = \{A_1, \ldots, A_n\}$, FDs F.
Repeat until X doesn't change;
do:
  if $\{B_1, \ldots, B_n\} \rightarrow C$ is in F
    and $\{B_1, \ldots, B_n\} \subseteq X$:
    then add C to X.
Return X as X^+

$\{name, category\}^+ =$
$\{name, category\}$

$\{name, category\}^+ =$
$\{name, category, color\}$

$\{name, category\}^+ =$
$\{name, category, color, dept\}$

F =
name → color
category → dept
color, category → price

$\{name, category\}^+ =$
$\{name, category, color, dept, price\}$

Exercise - 3

R(A,B,C,D,E,F)

A,B → C
A,D → E
B → D
A,F → B

Compute $\{A,B\}^+ = \{A, B, \quad\}$

Compute $\{A, F\}^+ = \{A, F, \quad\}$

Exercise - 4

- Find all FD's implied by

A,B → C
A,D → B
B → D

Requirements

1. **Non-trivial** FD (i.e., no need to return A, B → A)
2. The right-hand side contains **a single** attribute (i.e., no need to return A, B → C, D)

Review

1. Functional Dependency (FD)
   - What is an FD?
2. Inference Problem
   - Whether or not a set of FDs imply another FD?
3. Closure
   - How to compute the closure of attributes?

Fix Anomalies: High-level Idea

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Student	Course
Mike	5003
Mary	5003
Sam	5003
..	..

Course	Room
5003	LT-5
8005	7303

Two Steps

1. Search for "bad" FDs in the table
2. *Keep decomposing the table into sub-tables* until no more bad FDs

Like a debugging process

Fix Anomalies: Outline

- "Good" vs. "Bad" FDs
- Boyce-Codd Normal Form
- Decompositions

"Good" vs. "Bad" FDs

EmpID	Name	Phone	Position
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

EmpID → Name, Phone, Position

Good FD since EmpID can determine everything

EmpID is a Key

Position → Phone

Bad FD since Phone cannot determine everything

Exercise

(Student, Course) is the key

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Student, Course → Room Good FD!

Course → Room Bad FD!

What's wrong with "Bad" FDs

- If X →Y is a Bad FD, then X functionally determines *some* of the attributes; therefore, those other attributes can be duplicated
  - Recall: this means there is redundancy
  - And redundancy like this can lead to data anomalies!

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Outline

- "Good" vs. "Bad" FDs
- Boyce-Codd Normal Form
- Decompositions

Boyce-Codd Normal Form (BCNF)

- Main idea is that we define "good" and "bad" FDs as follows:
  - X → A is a "*good FD" if X is a key*
    - In other words, if A is the set of all attributes
  - X → A is a "*bad FD*" otherwise
- We will try to eliminate the "bad" FDs!

Boyce-Codd Normal Form (BCNF)

A relation R is **in BCNF** if:
there are no "bad" FDs

A relation R is **in BCNF** if:
if **$\{A_1, ..., A_n\} \rightarrow B$** is a *non-trivial* FD in R
then **$\{A_1, ..., A_n\}$ is a key** for R

Equivalently: ∀ sets of attributes X, either (X^+ = X) or (X^+ = all attributes)

Example

Is this table in BCNF?

Name	SIN	PhoneNumber	City
Fred	123-45-6789	604-555-1234	Vancouver
Fred	123-45-6789	604-555-6543	Vancouver
Joe	987-65-4321	908-555-2121	Burnaby
Joe	987-65-4321	908-555-1234	Burnaby

{SIN} → {Name,City}

This FD is *bad* because it is **not** a key

⇒ **Not** in BCNF

What is the key? {PhoneNumber}

Example

Name	SIN	City
Fred	123-45-6789	Vancouver
Joe	987-65-4321	Burnaby

SIN	PhoneNumber
123-45-6789	604-555-1234
123-45-6789	604-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

{SIN} → {Name,City}

This FD is now *good* because it is the key

Now in BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):

BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

X is not a key, i.e., $X^+ \neq$ [all attributes]

BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

If no "bad" FDs found, in BCNF!

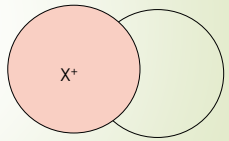
BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

Split R into X^+ and X+[rest attributes]

One table is X^+



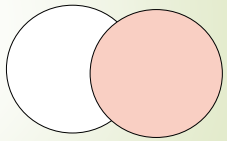
BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

Split R into X^+ and X+[rest attributes]

The other table is X + (R – X^+)



BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

Split R into X^+ and X+[rest attributes]

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no more "bad" FDs!

BCNF Decomposition Algorithm

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

Split R into X^+ and X+[rest attributes]

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Only look at the FD in the given set

Need to imply all FDs for R_1 and R_2

Example

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Course → Room

Student	Course
Mike	5003
Mary	5003
Sam	5003
..	..

Course	Room
5003	LT-5
8005	7303

Exercise - 2

BCNFDecomp(R):
Find *a non-trivial bad FD: X → Y*

if (not found) **then Return** R

Split R into X^+ and X+[rest attributes]

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

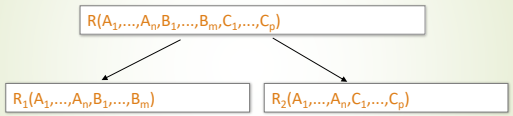
R(A,B,C,D,E)

{A} → {B,C}
{C} → {D}

Outline

- "Good" vs. "Bad" FDs
- Boyce-Codd Normal Form
- **Decompositions**

Decompositions in General



R_1 = the *projection* of R on $A_1, ..., A_n, B_1, ..., B_m$
R_2 = the *projection* of R on $A_1, ..., A_n, C_1, ..., C_p$

Lossless Decompositions

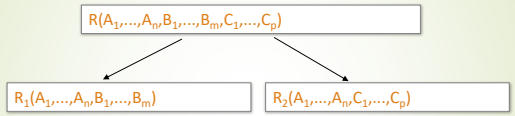
Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

Name	Price
Gizmo	19.99
OneClick	24.99
~~Gizmo~~	~~19.99~~

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

It is a **Lossless decomposition**

Lossless Decompositions



A decomposition R to (R1, R2) is **lossless** if R = R1 Join R2

Lossy Decomposition

Name	Price	Category
Gizmo	19.99	Gadget
OneClick	24.99	Camera
Gizmo	19.99	Camera

However sometimes it isn't

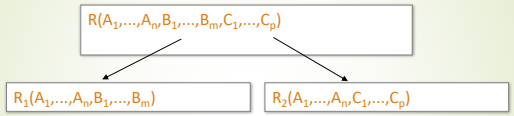
What's wrong here?

Name	Category
Gizmo	Gadget
OneClick	Camera
Gizmo	Camera

Price	Category
19.99	Gadget
24.99	Camera
19.99	Camera

Lossless Decompositions

Lossless decomposition: $R = R_1 \bowtie R_2$



If $\{A_1, ..., A_n\} \rightarrow \{B_1, ..., B_m\}$
Then the decomposition is lossless

Note: don't need $\{A_1, ..., A_n\} \rightarrow \{C_1, ..., C_p\}$

BCNF decomposition is always lossless. Why?

A Problem with BCNF

Unit	Company	Product
…	…	…

{Unit} → {Company}
{Company,Product} → {Unit}

Unit	Company
…	…

{Unit} → {Company}

Unit	Product
…	…

We do a BCNF decomposition on a "bad" FD:
$\{Unit\}^+$ = {Unit, Company}

We lose the FD {Company,Product} → {Unit}!!

The Problem

- We started with a table R and FDs F
- We decomposed R into BCNF tables $R_1, R_2, ...$ with their own FDs $F_1, F_2, ...$
- We insert some tuples into each of the relations—which satisfy their local FDs but when reconstruct it violates some FD **across** tables!

Practical Problem: To enforce FD, must reconstruct R—*on each insertion!*

Trade-offs

- **Different Normal Forms**

Prevent Decomposition Problems

VS

Remove Redundancy

BCNF still most common- with additional steps to keep track of lost FDs…

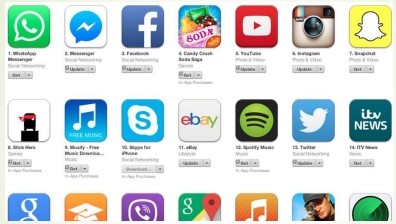
SDSC5003

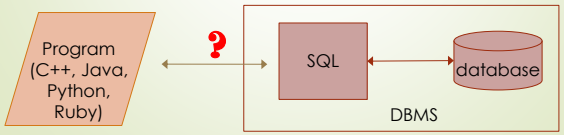
Database Application

Instructor: Yu Yang

yuyang@cityu.edu.hk

Programming Environment





Summary

- Normal Forms
  - 1st Normal Form (1NF)
  - 2nd Normal Form (2NF)
  - Boyce-Codd Normal Form (BCNF)
- Functional Dependency
  - Closure
  - Good and Bad FDs
- Boyce-Codd Normal Form
- Decompositions

Why this lecture

- **DB designer:** establishes schema
- **DB administrator:** tunes systems and keeps whole things running
- **Data scientist:** manipulates data to extract insights
- **Data engineer:** builds a data-processing pipeline
- **DB application developer:** writes programs that query and modify a database

Connecting to DBMS

- Fully embed into language (embedded SQL)
- Low-level library with core database calls (DB API)
- Stored Procedures
- Object-relational mapping (ORM)
  - Ruby on rails, django, etc
  - define database-backed classes
  - magically maps between database rows & objects
  - magic is a double-edged sword

Readings

- **RG Chapter 19**
- GWU Chapter 3

Outline

- **Database Programming**
- Application Architecture

Connecting to DBMS

- **Fully embed into language (embedded SQL)**
- Low-level library with core database calls (DB API)
- Stored Procedures
- Object-relational mapping (ORM)
  - Ruby on rails, django, etc
  - define database-backed classes
  - magically maps between database rows & objects
  - magic is a double-edged sword

Embedded SQL

- **Extend host language (CPP) with SQL syntax**

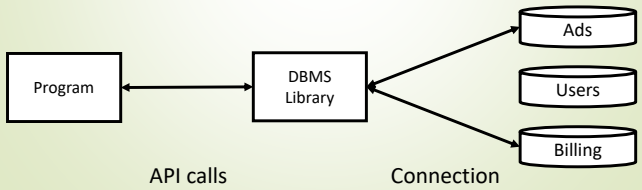
```
4  int main() {
5    EXEC SQL INCLUDE SQLCA;
6    EXEC SQL BEGIN DECLARE SECTION;
7      int OrderID;        /* Employee ID (from user)       */
8      int CustID;         /* Retrieved customer ID         */
9      char SalesPerson[10]  /* Retrieved salesperson name  */
10     char Status[6];     /* Retrieved order status        */
11   EXEC SQL END DECLARE SECTION;
12
13   /* Prompt the user for order number */
14   printf ("Enter order number: ");
15   scanf_s("%d", &OrderID);
16
17   /* Execute the SQL query */
18   EXEC SQL SELECT CustID, SalesPerson, Status
19     FROM Orders
20     WHERE OrderID = :OrderID
21     INTO :CustID, :SalesPerson, :Status;
22
23   /* Display the results */
24   printf ("Customer number:  %d\n", CustID);
25   printf ("Salesperson: %s\n", SalesPerson);
26   printf ("Status: %s\n", Status);
27   exit();
28 }
```

Declaring Variables

Embedded SQL Query

What does a library need to do?

- Single interface to possibly multiple DBMS engines
- Connect to a database
- Map objects between host language and DBMS
- Manage query results

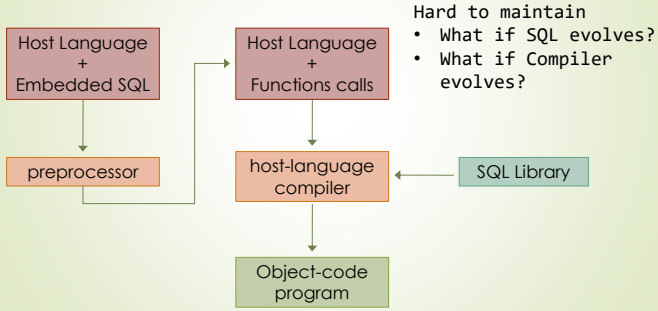


Query Execution

```
foo = conn.execute("select * from student")
```

- Challenges
  - Type Mismatch
  - What is the return type of `execute()`?
  - How to pass data between DBMS and host language?

Embedded SQL



ODBC and JDBC

- ODBC (Open DataBase Connectivity)

ODBC was originally developed by Microsoft and Simba Technologies

- JDBC (Java DataBase Connectivity)
  - Sun developed as set of Java interfaces
    - javax.sql.*

Type Mismatch

- SQL standard defines mappings between SQL and several languages

SQL types	C types	Python types
CHAR(20)	char[20]	str
INTEGER	int	int
SMALLINT	short	int
REAL	float	float

Connecting to DBMS

- Fully embed into language (embedded SQL)
- **Low-level library with core database calls (DB API)**
- Stored Procedures
- Object-relational mapping (ORM)
  - Ruby on rails, django, etc
  - define database-backed classes
  - magically maps between database rows & objects
  - magic is a double-edged sword

Connections

- Create a connection
  - Allocate resources for the connection
  - Relatively expensive to set up, libraries often cache connections for future use

```
conn = connect(sdsc5003.db)
```

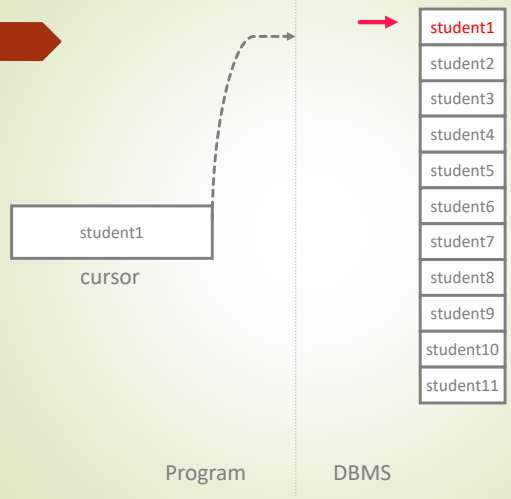
Should close connections when done! Otherwise resource leak.

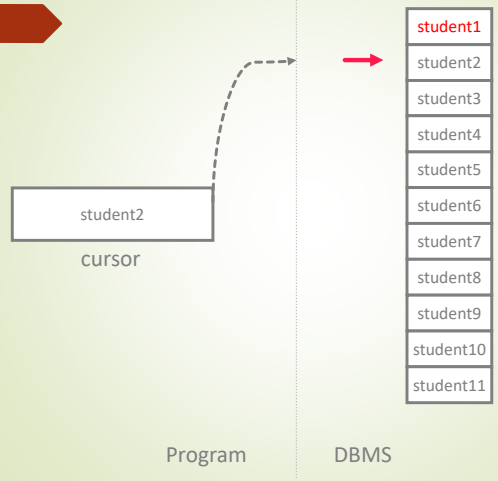
Cursor

- SQL relations and results are sets of records
- What is the type of `foo`?

```
foo = conn1.execute("select * from student")
```

- Cursor over the Result Set
  - similar to an iterator interface
  - Note: relations are unordered!
    - Cursors have no ordering guarantees
    - Use ORDER BY to ensure an ordering





Cursor

- Cursor similar to an iterator
  - cursor = conn.execute("SELECT * FROM student")
- Cursor methods
  - fetchone()
  - fetchall()

Cursor

- Cursor similar to an iterator

```
In [12]: import sqlite3
         conn = sqlite3.connect('C:/Users/yuyang/Desktop/A2.db')
         def search(name, conn):
             cursor = conn.execute("select E.eid,E.ename,E.age,E.salary \
                                    from Dept D, Works W, Emp E \
                                    where D.did=W.did and W.eid=E.eid and D.dname=?", \
                                    (name,))
             for record in cursor.fetchall():
                 print(record)
             conn.close()

         search('Hardware', conn)

(242518965, 'James Smith', 68, 27099.0)
(141582651, 'Mary Johnson', 44, 94011.0)
(141582657, 'Stanley Browne', 23, 14093.0)
(619023588, 'Jennifer Thomas', 24, 34654.0)
```

SQL Injection!!

symbol = "RHAT' OR True -- "

SELECT * **FROM** stocks **WHERE** symbol = 'RHAT' OR True -- '

```
are Foundation [US] | https://docs.python.org/2/library/sqlite3.html

# Never do this -- insecure!
symbol = 'RHAT'
c.execute("SELECT * FROM stocks WHERE symbol = '%s'" % symbol)

# Do this instead
t = ('RHAT',)
c.execute('SELECT * FROM stocks WHERE symbol=?', t)
print c.fetchone()
```

SELECT * **FROM** stocks **WHERE** symbol = 'RHAT'' OR True -- '

Connecting to DBMS

- Fully embed into language (embedded SQL)
- Low-level library with core database calls (DB API)
- **Stored Procedures**
- Object-relational mapping (ORM)
  - Ruby on rails, django, etc
  - define database-backed classes
  - magically maps between database rows & objects
  - magic is a double-edged sword

Stored Procedures

- A *stored procedure* is a function / procedure written in a general-purpose programming language that is executed within the DBS.
- Performs computations that cannot be expressed in SQL.
- Procedure executed through a single SQL statement.
- Executed in the process space of the DB server.
- SQL standard: *PSM* (Persistent Stored Modules). Extends SQL by basic concepts of a general-purpose programming language.

Benefits of Stored Procedures

- Stored procedures are modular
  - It is easier to change a stored procedure than to edit an embedded query
  - This makes it easier to maintain stored procedures, and to
  - Change the procedure to increase its efficiency
- Stored procedures are registered with the DB server
  - They can be used by multiple applications and
  - Avoid tuple-at-a-time return of records through cursors
  - Separate server-side functions from client-side functions

Stored Procedures: Examples

```
CREATE PROCEDURE ShowNumReservations
    SELECT S.sid, S.sname, COUNT(*)
    FROM Sailors S, Reserves R
    WHERE S.sid = R.sid
    GROUP BY S.sid, S.sname
```

Stored procedures can have parameters:

- Three different modes: IN, OUT, INOUT

```
CREATE PROCEDURE IncreaseRating(
    IN sailor_sid INTEGER, IN increase INTEGER)
    UPDATE Sailors
    SET rating = rating + increase
    WHERE sid = sailor_sid
```

Stored Procedures: Examples (Contd.)

Stored procedure do not have to be written in SQL:

```
CREATE PROCEDURE TopSailors(
   IN num INTEGER)
LANGUAGE JAVA
EXTERNAL NAME "file:///c:/storedProcs/rank.jar"
```

Main SQL/PSM Constructs (Contd.)

- Local variables (DECLARE)
- RETURN values for FUNCTION
- Assign variables with SET
- Branches and loops:
  - IF (condition) THEN statements;
    ELSEIF (condition) statements;
    … ELSE statements; END IF;
  - LOOP statements; END LOOP
- Queries can be parts of expressions
- Can use cursors without "EXEC SQL"

Calling Stored Procedures

```
EXEC SQL BEGIN DECLARE SECTION
Int sid;
Int rating;
EXEC SQL END DECLARE SECTION

// now increase the rating of this sailor
EXEC SQL CALL IncreaseRating(:sid,:rating);
```

SQL/PSM

Most DBMSs allow users to write stored procedures in a simple, general-purpose language (close to SQL) → SQL/PSM standard is a representative

Declare a stored procedure:

```
CREATE PROCEDURE name(p1, p2, …, pn)
   local variable declarations
   procedure code;
```

Declare a function:

```
CREATE FUNCTION name (p1, …, pn) RETURNS
   sqlDataType
   local variable declarations
   function code;
```

Main SQL/PSM Constructs

```
CREATE FUNCTION rate Sailor
      (IN sailorId INTEGER)
      RETURNS INTEGER
DECLARE rating INTEGER
DECLARE numRes INTEGER
SET numRes = (SELECT COUNT(*)
                  FROM Reserves R
                  WHERE R.sid = sailorId)
IF (numRes > 10) THEN rating =1;
ELSE rating = 0;
END IF;
RETURN rating;
```

SQL Server Version

```
CREATE FUNCTION rateSailor (@sailorId INT)
  RETURNS INT
AS
BEGIN
  DECLARE @numRes INT
  DECLARE @rating INT
      SET @numRes = (SELECT COUNT(*)
            FROM Reserves R
            WHERE R.sid = @sailorId)
  IF @numRes > 10
    SET @rating = 1
  ELSE
    SET @rating = 0
  RETURN @rating
END
GO;
SELECT dbo.rateSailor(22); go
```

Outline

- Database Programming
- **Application Architecture**

Application Architectures

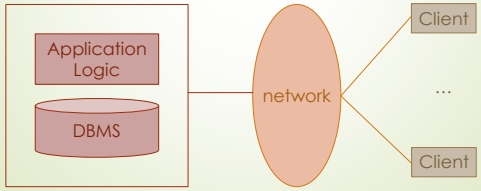
- Single tier
  - How things used to be …
- Two tier
  - Client-server architecture
- Three tier (and multi-tier)
  - Used for many web systems
  - Very scalable

Single Tier Architecture

- Historically, data intensive applications ran on a single tier which contained
  - The DBMS,
  - Application logic and business rules, and
  - User interface

Two-Tier Architecture

- Client/ server architecture
  - The server implements the business logic and data management
- Separate presentation from the rest of the application

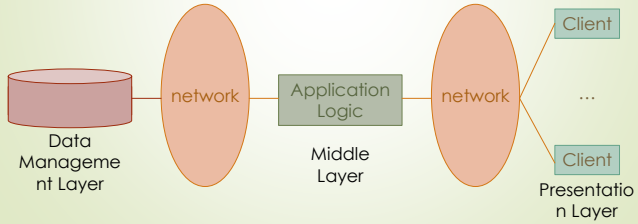


Presentation Layer

- Responsible for handling the user's interaction with the middle tier
- One application may have multiple versions that correspond to different interfaces
  - Web browsers, mobile phones, …
  - Style sheets can assist in controlling versions

Three-Tier Architecture

- Separate presentation from the rest of the application
- Separate the application logic from the data management



Business logic Layer

- The middle layer is responsible for running the business logic of the application which controls
  - What data is required before an action is performed
  - The control flow of multi-stage actions
  - Access to the database layer
- Multi-stage actions performed by the middle tier may require database access
  - But will not usually make permanent changes until the end of the process
    - e.g. adding items to a shopping basket in an Internet shopping site

Data Management Layer

- The data management tier contains one, or more databases
  - Which may be running on different DBMSs
- Data needs to be exchanged between the middle tier and the database servers
  - This task is not required if a single data source is used but,
  - May be required if multiple data sources are to be integrated
  - *XML* is a language which can be used as a data exchange format between database servers and the middle tier

Example: Airline reservations

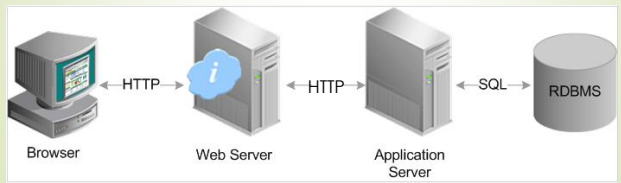
- Consider the three tiers in a system for airline reservations
- Database System
  - Airline info, available seats, customer info, etc.
- Application Server
  - Logic to make reservations, cancel reservations, add new airlines, etc.
- Client Program
  - Log in different users, display forms and human-readable output

Example: Course Enrollment

- Student enrollment system tiers
- Database System
  - Student information, course information, instructor information, course availability, pre-requisites, etc.
- Application Server
  - Logic to add a course, drop a course, create a new course, etc.
- Client Program
  - Log in different users (students, staff, faculty), display forms and human-readable output

3 Tier Architecture and the Web

- In the domain of web applications three tier architecture usually refers to
  - Web server
  - Application server
  - Database server



Summary

- Database Programming
  - Embedded SQL
  - DB API
- Application Architecture
  - Three Tier Architecture

Readings

- **RG Chapter 6, Chapter 7**
- GWU Chapter 9

SDSC5003

Intro to Time Complexity & Basic Indexing Structures

Instructor: Yu Yang

yuyang@cityu.edu.hk

Why this lecture

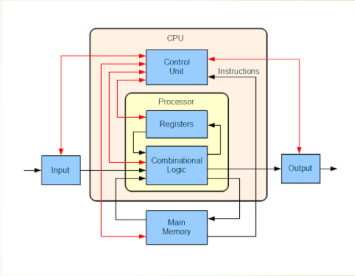
- Most of you do not have CS background
- Help you better understand the efficiency issue of Data Store and Retrieval
- Computation as an important aspect of Data Science
  - Math: there exist a solution
  - Statistics: data reveals the underlying mechanism
  - Engineering: model real applications as data problems
  - **Computation**: efficiently construct the solution
    - Time complexity: measure of efficiency

Outline

- **Intro to Time Complexity**
- Basic Data Structures

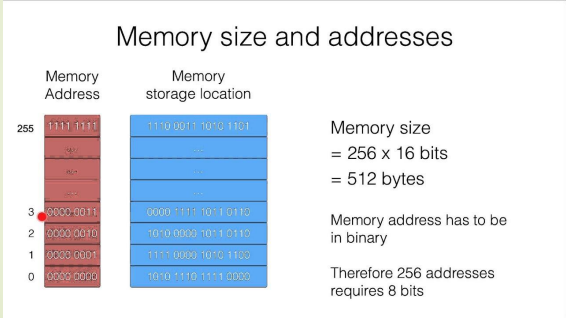
Modern PC Architecture

CPU
- Logic/Arithmetic operations (basic computational step)
- FLOPS: floating point operations per second
- CPU Frequency



Main Memory vs I/O
- Memory is fast, random access, small storage
- I/O is slow, sequential access, large storage

Main Memory



Similar to Array A[i]

Time Complexity

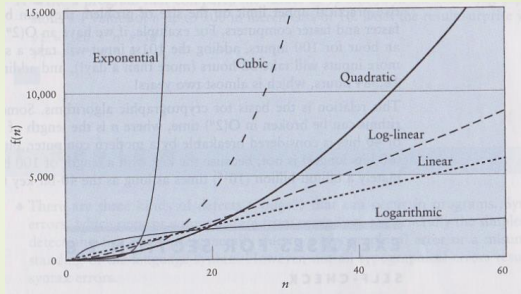
- Resources are limited: time, storage, etc.
- Measures how many computational steps in relation to input
  - As input size increases, how much impact on computation time?
- Space Complexity: Similar to Time Complexity

Big O Notation

- Input size: number of bits needed, denote by n
- Time Complexity O(f(n))
  - #Steps, **worst case**
  - #Steps ≤ Cf(n), C is a constant (finite)

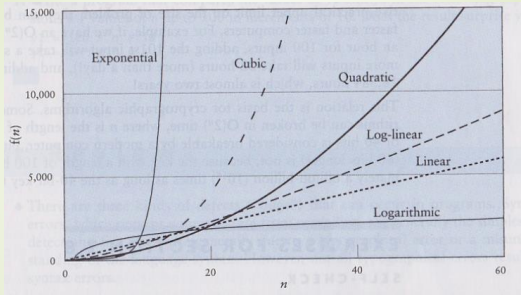
- **O(1) – Constant.** Very Nice!
- **O(log n) – Logarithmic.** Nice!
- **O(n) – Linear.** Good!
- **O(n log n) – Log-linear.** Not Bad.
- **O(n^2) – Quadratic.** Getting expensive.
- **O(n^3) – Cubic.** Expensive.
- **O(2^n) – Exponential.** Ouch!
- **O(n!) – Factorial.** Holy Moly!!

Scalability



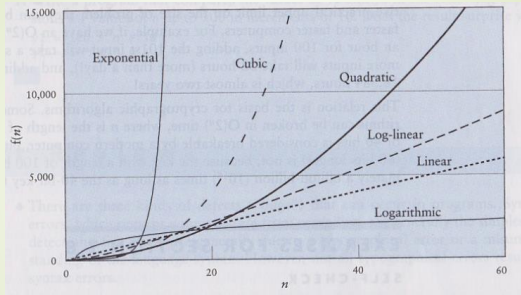
Enumerating all subsets of {1,2,…,200}
How big is 2^{200} ?

Scalability



Enumerating all subsets of {1,2,…,200}
How big is 2^{200} ?

Scalability



Enumerating all subsets of {1,2,…,200}
How big is 2^{200} ?
Age of universe≈4.3×10^{17} s , IBM Summit: 2×10^{17} Flops ,
$4.3\times10^{17}\times2\times10^{17} \ll 1.6\times10^{60}\approx2^{200}$

O(1) Example

```
public void setNum(int location, int[] A, int num)
{
    A[location] = num;
}
```

Remember that we have random access to A[]

O(n) Example

```
public int findMax(int[] A)
{
    int n = A.length;
    int max_val = Integer.MIN_VALUE;
    for (int i = 0; i < n; ++i)
    {
        if (A[i] > max_val)
            max_val = A[i];
    }
    return max_val;
}
```

O(n^2) Example

```
public void sortAsc(int[] A)
{
    int n = A.length;
    for (int i = 0; i < n; ++i)
    {
        for(int j = i; j < n; ++j)
        {
            if (A[j] < A[i])
            {
                swap(A[i], A[j]); // tmp=A[i], A[j]=A[i], A[i]=tmp
            }
        }
    }
}
```

Exercise: What complexity?

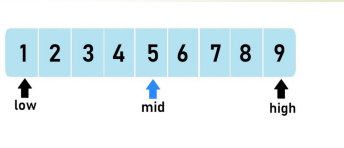
```
public boolean isPrime(int n)
{
    for(int i = Math.sqrt(n); i >= 2; --i)
    {
        if(n % i == 0)
            return false;
    }
    return true;
}
```

Outline

- Intro to Time Complexity
- **Basic Data Structures**

A First Bite: Binary Search

- Suppose array A is sorted in ascending order
- We want to find if num is in A
  - low = 0, high = A.length-1
  - mid = (low+high)/2
  - Check if A[mid] == num
  - A[mid] > num: high = mid-1
  - A[mid] < num: low = mid+1
  - Terminate if low > high



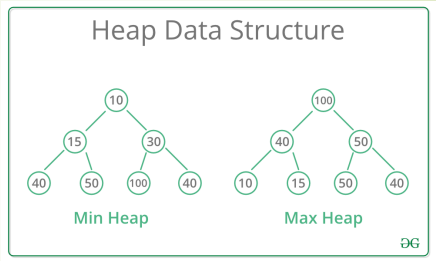
- O(log n)

Sorted Array

- Query complexity O(log n)
- Issue: maintenance
  - Insertion: O(n) (Why?)
  - Deletion: O(n)

Heap

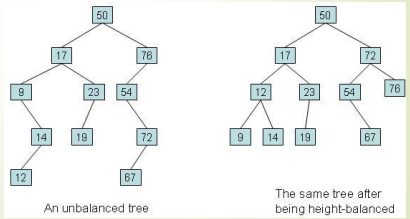
- A complete binary tree
  - Fully filled except for the bottom level
- Every node is no greater/smaller than its children
- Max/Min Query: O(1)
- Root deletion: O(log n)
- Insertion: O(log n)



(Balanced) Binary Search Tree

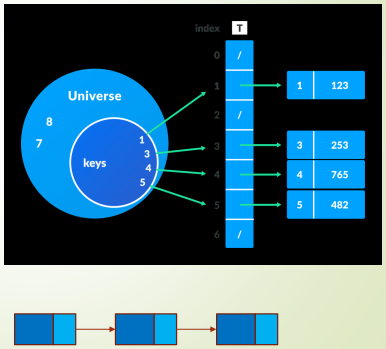
- A binary tree
  - | Height(left tree)-Height(right tree) | <=1
  - Height?
- Left child <= node <= right child
- Query: O(log n)
- Insertion: O(log n)
- Deletion: O(log n)
- Iteration: O(n)

Height is important in tree structures!



Random DS: Hash Table

- Hash function H: D->N
  - x MOD p, p is a large prime number
  - May have conflicts
- Hash Table: O(n) Array of lists
  - Query: O(1)
  - Insertion: O(1)
  - Deletion: O(1)
  - All amortized complexity
- A list often is not continuous in Main Memory
  - X, pointer next, pointer pre
  - No random access



Readings

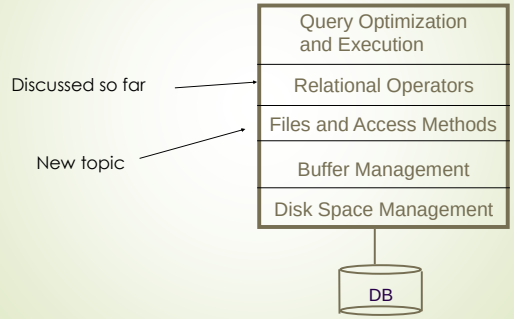
- Any introductory book/slides/tutorial on data structures & algorithm design
- **Data Structures and Algorithms. Aho, Ullman & Hopcroft**

SDSC 5003 Overview of Storage and Indexing

Instructor: Yu Yang

yuyang@cityu.edu.hk

System Issues: How to Build a DBMS



Data on External Storage

- Disks: Can retrieve random page at fixed cost
  - But reading several consecutive pages is much cheaper than reading them in random order
- Tapes: Can read pages only in sequence
  - Cheaper than disks; used for archival storage
- File organization: Method of arranging a file of records on external storage.
  - Record id (rid) is sufficient to physically locate record
  - 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 gets 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, but 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 has some structures and can be organized 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.

How about relations?

- Relation is typically stored as a file of records (= tuples)
- A file corresponds to several pages
- Page size is typically 4KB – 8KB.
- If use Sorted files: can sort on only one order.

Indexes

Indexes

- 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 (e.g., age or color).
  - *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 search key value **k**.

Alternatives for Data Entry **k*** in Index

- Three alternatives:
  - Data record with search key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key value **k**>

Example of Alternative 1

Loca-tion	shape	color	holes
1	round	Red	2
2	square	Red	4
3	rectangle	Red	8
4	round	blue	2
5	square	blue	4
6	rectangle	blue	8

6 data entries, sorted by color

Alternatives for Data Entries (Contd.)

- Alternative 1:
  - If this is used, index structure is actually a special file organization (like a Heap file or sorted file).
  - At most one index on a given collection of data records can use Alternative 1.
    - 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.
    - Otherwise, data records are duplicated

Example of Alternative 2

Loca-tion	color
1	Red
2	Red
3	Red
4	blue
5	blue
6	blue

6 data entries, sorted by color

Example of Alternative 3

Loca-tions	color
1, 2, 3	Red
4,5,6	Blue

2 data entries, variable length

Alternatives for Data Entries (Contd.)

- Alternatives 2 and 3:
  - Data entries typically much smaller than data records.
    - So, better than Alternative 1 with large data records.
  - Alternative 3 more compact than Alternative 2.

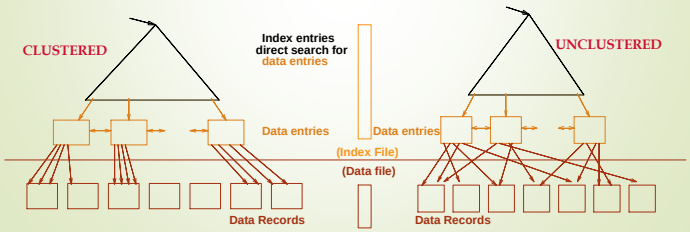
Index Types

Index Classification

- *Primary* vs. *secondary*: If search key contains primary key, then called primary index.
  - *Unique* index: Search key uniquely identifies record.
- *Clustered* vs. *unclustered*: 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); strictly, not vise-versa.
  - 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 insertions. (Thus, order of data records is 'close to', but not identical to, the sort order.)



Example of Alternative 1 (cont.)

id	shape	color	holes
1	round	Red	2
2	square	Red	4
3	rectangle	Red	8
4	round	blue	2
5	square	blue	4
6	rectangle	blue	8

Suppose each page contain only 3 tuples.
Search key is color.
Data entries for will be Red, blue
How about clustered alternative 2?
How about unclustered alternative 2?

Hash-Based Indexes

- Good for equality selections.
  - Index is a collection of *buckets*. Bucket = *primary page* plus zero or more *overflow pages*.
  - *Hashing function* **h**: **h**(*r*) = bucket in which record *r* belongs. **h** looks at the *search key* fields of *r*.
- If Alternative (1) is used, the buckets contain the data records; otherwise, they contain <key, rid> or <key, rid-list> pairs.

An example

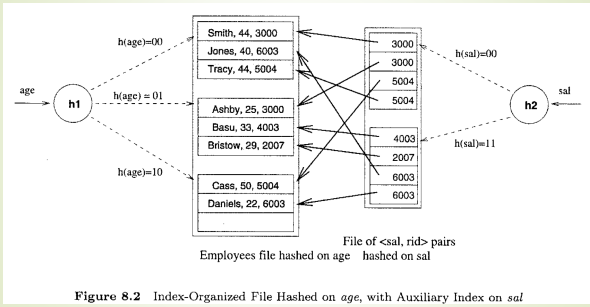
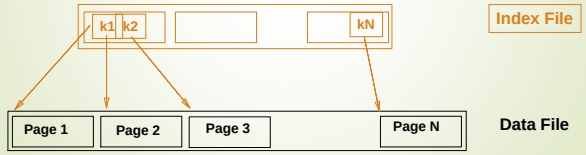


Figure 8.2 Index-Organized File Hashed on *age*, with Auxiliary Index on *sal*

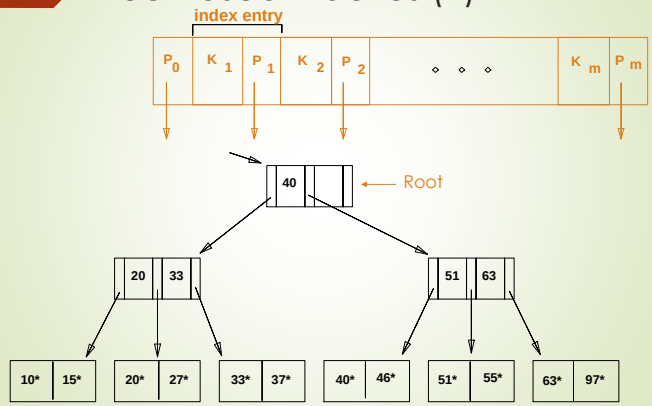
Tree-Based Indexes

- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



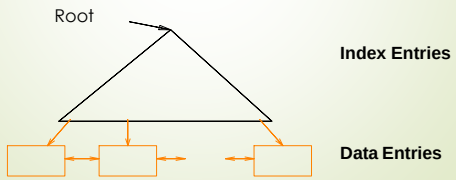
- *Can do binary search on (smaller) index file!*

Tree-Based Indexes (2)



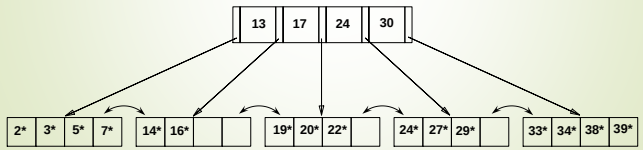
B+ Tree: The Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf.
- Search for 5*, 15*, all data entries >= 24* ...



B+ Trees in Practice

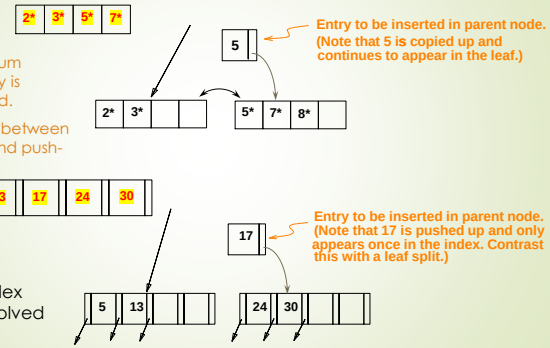
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Inserting a Data Entry into a B+ Tree

- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)

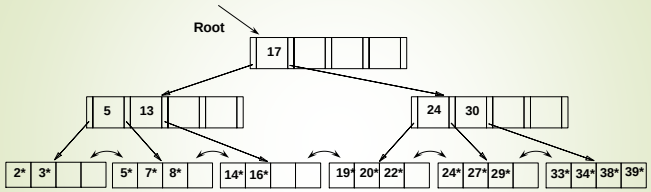
Inserting 8* into Example B+ Tree

- Note:
  - why minimum occupancy is guaranteed.
  - Difference between copy-up and push-up.



How many index pages are involved in insert?

Example B+ Tree After Inserting 8*

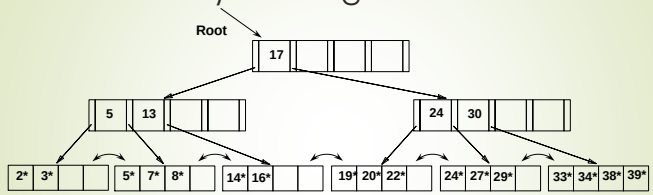


Notice that root was split, leading to increase in height.

Deleting a Data Entry from a B+ Tree

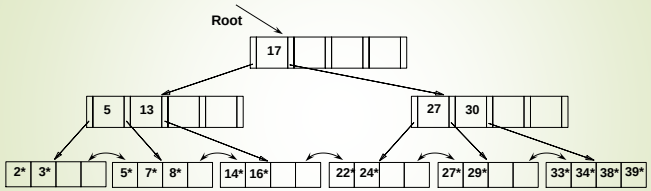
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* L and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

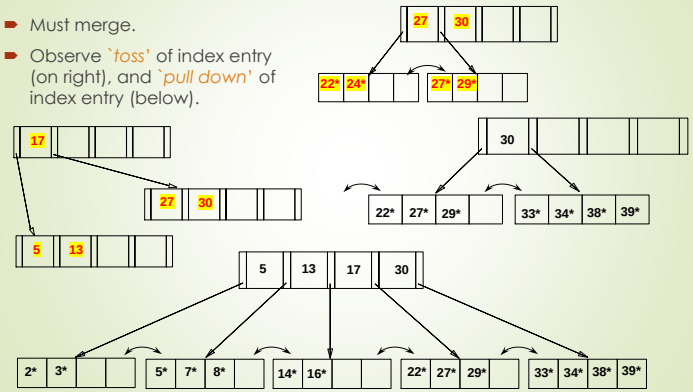
Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).



Efficiency Analysis

When to use what index

Cost Model for Our Analysis

Notations:

- **B:** The number of data pages
- **R:** Number of records per page
- **H:** time to apply the hash function
- **D:** (Average) time to read or write disk page
- **C:** (Average) time to process a record (e.g., to compare a field value to a selection constant)
- Average-case analysis; based on several simplistic assumptions.

 Good enough to show the overall trends!

Assumptions in Our Analysis

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
  - Clustered files: pages typically 67% full.
  - ⇒ Total number pages needed = 1.5 B.
- (Unclustered) Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy.
    - ⇒ Index size = 1.25 B data size.
    - ⇒ #data entries/page = 10 (0.8R) = 8R.
  - Tree: 67% page occupancy of index pages (this is typical).
    - ⇒ #leaf pages = (1.5 B) 0.1 = 0.15 B.
    - ⇒ #data entries/page = 10 (0.67R) = 6.7R.

Exercise for Group Work

1. Estimate how long an equality search takes in (i) a heap file (ii) a sorted file (iii) a hash file, hashed on the search key, with at most one record matching the search key (i.e., the search is on a key field).

2. Estimate how long an insertion takes in (i) a heap file (ii) a sorted file (iii) a hash file. Assume that insertion in a heap file is at the end, and that the sorted file has no empty slots.

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 msec	100 nanosec	100 nanosec	100

The I/O cost dominates

- Efficiency gap between memory and disk
- And,

Parameters of the Analysis Mlsec=10^{-3}s Nanosec=10^{-9}s

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 mlsec	100 nanosec	100 nanosec	100

Operations to Compare

- Scan: Fetch all records from disk
- Equality search (e.g., "age = 30")
- Range selection (e.g., "age > 30")
- Insert a record
- Delete a record

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

 Several assumptions underlie these (rough) estimates!
 Order of magnitude results, omit R,C, H.

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files
- Clustered B+ tree file, Alternative (1)
- Heap file with unclustered B + tree index
- Heap file with unclustered hash index
- NOTE indexes are on <age,sal>

Scanning Cost

- Heap file: B(D + RC).
  - for each page (B)
  - Read the page (D)
  - For each record (R), process the record (C).
- Sorted File: B(D + RC).
  - Have to go through all pages.
- Clustered File: 1.5B (D+RC).
  - Pages only 67% full.
- Unclustered Tree Index: >BR(D+C). Bad!
  - for each record (BR)
  - retrieve page and find record (D + C).

Selecting Indexes

Create Indexes in SQL

- SQL supports many options for creating index (more than we can cover).
- Sample Syntax:

  create index IX_Product_Color

  on Product (Color);
- More examples:

  CREATE [UNIQUE] INDEX index_name
  ON table_name (column_name)

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?

Choice of Indexes (Contd.)

- 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.

Examples of Clustered Indexes

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have *E.age* > 10, using *E.age* index and sorting the retrieved tuples may be costly.
  - Clustered *E.dno* index may be better!
- Equality queries and duplicates:
  - Clustering on *E.hobby* helps!

```
SELECT E.dno
FROM Emp E
WHERE E.age>40
```

```
SELECT E.dno, COUNT (*)
FROM Emp E
WHERE E.age>10
GROUP BY E.dno
```

```
SELECT E.dno
FROM Emp E
WHERE E.hobby='Stamps'
```

Indexes with Composite Search Keys

- 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.

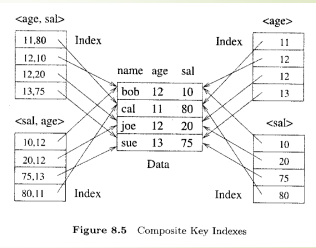


Figure 8.5 Composite Key Indexes

Index-Only Plans

With a suitable index, some queries can be answered without retrieving any tuples from one or more relations involved

<*E.dno,E.eid*>
Tree index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<*E.dno*>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<*E.dno,E.sal*>
Tree index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<*E. age,E.sal*>
or
<*E.sal, E.age*>
Tree!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

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!
- Try to choose indexes that benefit as many queries as possible.
  - Since only one index can be clustered per relation, choose it based on important queries that would benefit the most from clustering.

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.

Summary (Contd.)

- 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, and primary vs. secondary.
  - Differences have important consequences for utility/performance.

Summary (Contd.)

- Understanding the nature of the *workload* for the application, and the performance goals, is essential to developing a good design.
  - What are the important queries and updates? What attributes/relations are involved?
- Indexes must be chosen to speed up important queries (and perhaps some updates!).
  - Index maintenance overhead on updates to key fields.
  - Choose indexes that can help many queries, if possible.
  - Build indexes to support index-only strategies.
  - Clustering is an important decision, demanding on DBMS but potentially high payoff.

Readings

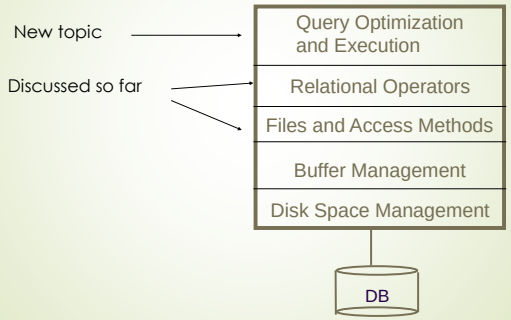
- **RG Chapter 8, Chapter 9**
- GUW Chapter 13, Chapter 14

Overview of Implementing Relational Operators and Query Evaluation

Instructor: Yu Yang

yuyang@cityu.edu.hk

System Issues: How to Build a DBMS



Motivation: Evaluating Queries

- The same query can be evaluated in different ways.
- The evaluation strategy (plan) can make orders of magnitude of difference.
- Query efficiency is one of the main areas where DBMS systems compete with each other.
- Decades of development, secret details.

Overview of Query Evaluation

- ***Plan:*** *Tree of Relational Algebra operators, with choice of algorithms for each operator*
  - Each operator typically implemented using a *pull interface*: when an operator is 'pulled' for the next output tuples, it 'pulls' on its inputs and computes them.
  - Similar as cursor/iterator
- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: find best plan.
- Practically: Avoid worst plans!

Show query execution plan in SQL Server

- Tick "Include Actual Execution Plan" menu item (found under the "Query" menu)
- When your query completes you should see an extra tab entitled "Execution plan" appear in the results pane as
- Example: http://stackoverflow.com/questions/7359702/how-do-i-obtain-a-query-execution-plan

Some Common Techniques

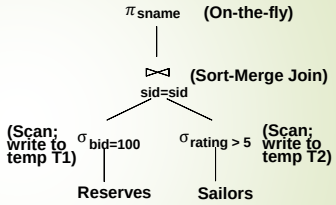
- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions and indexes to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing on a sort key, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

** Watch for these techniques as we discuss query evaluation!*

Examples

Alternative Plan

- *Goal of optimization:* To find efficient plans that compute the same answer.



Exercise 12.4

Consider the following schema with the Sailors relation:

Sailors(sid: integer, sname: string, rating: integer, age: real)

- For each of the following indexes, list whether the index matches the given selection conditions.

1. A hash index on the search key <Sailors.sid>
   a. $\sigma_{sid<50,000}$ (Sailors)
   b. $\sigma_{sid=50,000}$ (Sailors)
2. A B+-tree on the search key <Sailors.sid>
   a. $\sigma_{sid<50,000}$ (Sailors)
   b. $\sigma_{sid=50,000}$ (Sailors)

Example Relations

Sailors

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

Reservations

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Computing Relational Operators

Exercise 12.4

Consider the following schema with the Sailors relation:

Sailors(sid: integer, sname: string, rating: integer, age: real)

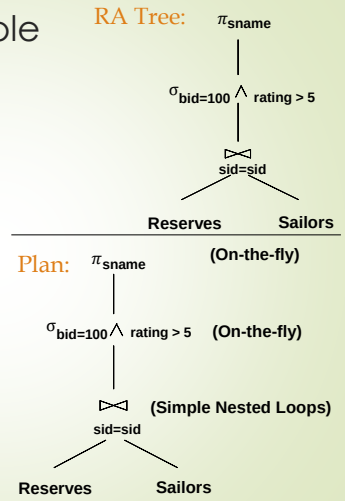
- For each of the following indexes, list whether the index matches the given selection conditions.

1. A hash index on the search key <Sailors.sid>
   a. $\sigma_{sid<50,000}$ (Sailors) No. Hash index cannot deal with inequalities
   b. $\sigma_{sid=50,000}$ (Sailors) Yes
2. A B+-tree on the search key <Sailors.sid>
   a. $\sigma_{sid<50,000}$ (Sailors) Yes
   b. $\sigma_{sid=50,000}$ (Sailors) Yes

Query Plan Example

```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

- RA Tree: expression tree.
- Each leaf is a schema table.
- Internal nodes: relational algebra operator applied to children.
- Full plan labels each internal node with implementation strategy.



Access Paths

- An access path is a method of retrieving tuples:
  - File scan, or index that matches a selection (in the query)
- A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
  - E.g., Tree index on <*a, b, c*> matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- A hash index *matches* (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.
  - E.g., Hash index on <*a, b, c*> matches *a=5 AND b=3 AND c=5*; but it does not match *b=3*, *or a=5 AND b=3*, *or a>5 AND b=3 AND c=5*.

Selection and Projection

One Approach to Selections

- Estimate the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that requires the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*
  - Other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
  - Consider *day<8/9/94 AND bid=5 AND sid=3*.
    - A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple.
    - Similarly, a hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked.

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
  - In example, assume that about 10% of tuples qualify a range selection(100 pages, 10000 tuples). With a clustered tree index, cost is little more than 100 I/Os; if unclustered, up to 10000 I/Os!

Projection

- *The expensive part is removing duplicates.*
  - SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.
  - For example, we want to the sid and bid from Reserves.
- Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.

The Biggie: Join

Nested Loops

Sort-Merge

Join: Index Nested Loops

Natural Join on Reserves and Sailors?

```
foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result
```

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: Pages_in_r * (1 + tup_per_page* cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index on S: 1 I/O (typical) for each R tuple, unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os for finding matches.
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 R/ 40,000 S). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

Exercise 14.4.1

- Consider the join R.a with S.b given the following information. The cost measure is the number of page I/Os, ignoring the cost of writing out the result.
  - Relation R contains 10,000 tuples and has 10 tuples per page.
  - Relation S contains 2000 tuples, also 10 tuples per page.
  - Attribute b is the primary key for S.
  - Both relations are stored as heap files. No indexes are available.
- What is the cost of joining R and S using nested loop join? R is the outer relation.
- How many tuples does the join of R and S produce, at most, and how many pages are required to store the result of the join back on disk?

Join: Sort-Merge ($R \bowtie_{i=j} S$)

- Sort R and S on the join column, then scan them to do a 'merge' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple.

Example of Sort-Merge Join

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

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: sort + scan = (M log M + N log N) + (M+N) [sid is key]
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With enough buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join I/O cost: 7500. (M=500, N=1000)

Exercise 14.4.3

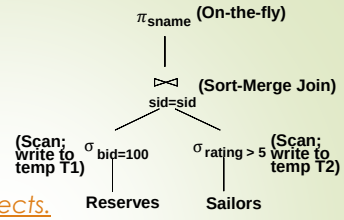
- Consider the join R.a with S.b given the following information. The cost measure is the number of page i/Os, ignoring the cost of writing out the result.
  - Relation R contains 10,000 tuples and has 10 tuples per page.
  - Relation S contains 2000 tuples, also 10 tuples per page.
  - Attribute b is the primary key for S.
  - Both relations are stored as heap files. No indexes are available.
- What is the cost of joining R and S using a sort-merge join? Assume that the number of I/Os for sorting a table T is 4 *pages_in_T .

Size Estimation and Reduction Factors

```
SELECT  attribute list
FROM    relation list
WHERE   term1 AND ... AND termk
```

- Consider a query block:
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.
  - Implicit assumption that *terms are independent!*
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Alternative Plans 1 (No Indexes)



- ***Main difference:*** *push selects.*
- With 5 buffer pages, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings, uniform distribution).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: (1000+10+500+250)+(40+2000+260) = 4060 page I/Os.

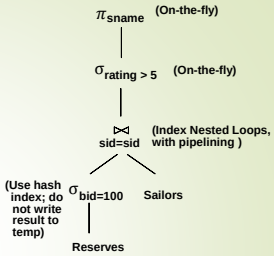
Query Planning

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Similar to old schema; *rname* added for variations.
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Alternative Plan 2 With Indexes



- An unclustered hash index on sid of Sailors
- And clustered index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages for selection.

- Join column *sid* is a key for Sailors.
  - ⇒ At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating>5* before the join:
  - there is an index on sid of Sailors, don't want to compute selection
- Cost: Selection of Reserves tuples (10 I/Os).
  - For each, must get matching Sailors tuple (1000*(1.2+1)).
  - Total 2210 page I/Os.

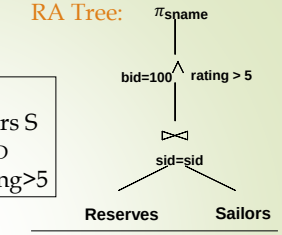
Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
    - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
  - Must also estimate *size of result* for each operation in plan tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.

Motivating Example

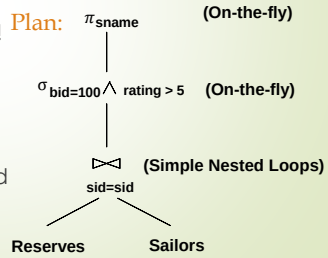
```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE   R.sid=S.sid AND
        R.bid=100 AND S.rating>5
```

RA Tree:



- Page based selection Cost: 1000+1000*500 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been 'pushed' earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.

Plan:



Summary

- There are several alternative evaluation algorithms for each relational operator.
- A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.
- Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Readings

- **RG Chapter 14, Chapter 15**
- GUW Chapter 15, Chapter 16

SDSC5003
Parallel Data Processing

Instructor: Yu Yang
yuyang@cityu.edu.hk

Outline

- Parallel RDBMS
- Intro to MapReduce
- Intro to Spark

Parallel RDBMS

Why Parallel Computing?

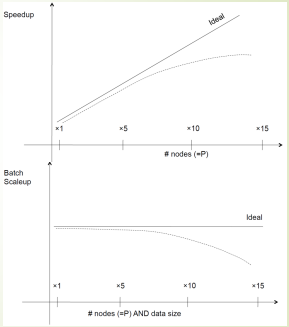
- Multicores
  - Most processors have multiple cores
- Big Data
  - Too big to fit in main memory
  - Distributed query processing on hundreds or thousands of servers
  - Widely available now using cloud services

Why Parallel Computing?

- Multicores
  - Most processors have multiple cores
- Big Data
  - Too big to fit in main memory
  - Distributed query processing on hundreds or thousands of servers
  - Widely available now using cloud services

Performance Metrics for Parallel RDBMS

- Nodes: processers, computers
- Speedup: more nodes, same data -> faster computation
- Scaleup: more nodes, more data -> same speed
- Why sub-linear speedup and scaleup?
  - Startup cost: cost of starting an operation on many nodes
  - Interference: resource contention among nodes
  - Skew: bottleneck is the slowest node

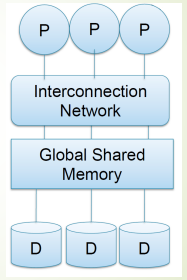


Architectures for Parallel DBMS

- Shared memory
- Shared disk
- Shared nothing

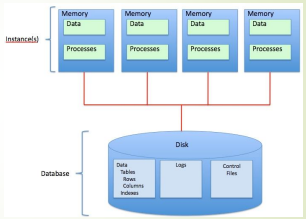
Shared Memory

- Nodes share both RAM and disk
- Dozens to hundreds of processers
- SQL Server runs on a single machine
  - Exploits many threads to speed up
- Easy to use and program
- Expensive to scale



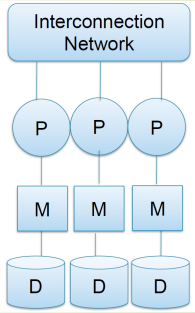
Shared Disk

- All nodes access the same disk
- Every node has its own memory
- Example: Oracle
- No need to worry about shared memory
- Hard to scale: existing deployments typically have fewer than 10 machines



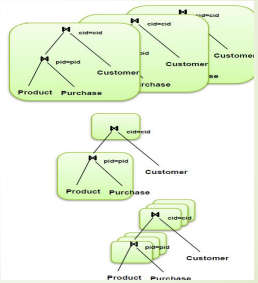
Shared Nothing

- Cluster of commodity machines on high-speed network (clusters)
- Every machine has its own memory and disk
- Example: Google
- Easy to maintain and scale
- Most difficult to tune



Approaches to Parallel Query Evaluation

- Inter-query parallelism
  - Transactions per node
  - Good for transactional workloads
- Inter-operator parallelism
  - Operator per node
  - Good for analytical workloads
- Intra-operator parallelism
  - Operator on multiple nodes

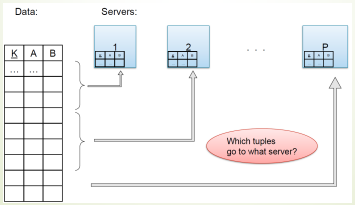


Single Node Query Processing

- Suppose we query relations R(A,B) and S(B,C) without indexes
- Selection: select * from R where R.A=1
  - Scan data file R, select records with A=1
- Group by: select A, sum(B) from R group by R.A
  - Scan data file R, insert into a hash table using A as key
  - When a new key equals an existing one, add B to the value
- Join: select * from R, S where R.B=S.B
  - Scan R, insert into a hash table using B as key
  - Scan S, probe the hash table using B

Distributed Query Processing

- Data is partitioned horizontally (by tuples) on many machines
- Operators may require data reshuffling
- A key problem: distributing data across multiple nodes



Horizontal Data Partitioning

- Block partition
  - Partition tuples arbitrarily s.t. size(R1)≈…≈size(Rp)
- Hash partition based on attribute A
  - Tuple t goes to chunk i if f(t.A)=I
- Range partition based on A
  - Partition the range of A into $-\infty = v_0 < v_1 < \cdots < v_p = \infty$
  - Tuple t goes to chunk i if $v_{i-1} < t.A < v_i$

Uniform Data vs Skewed Data

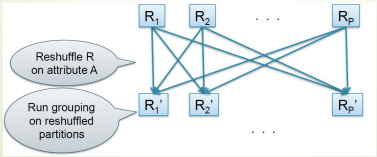
- R(K,A,B,C)
- Block partition: uniform
- Hash partition
  - On the key K: uniform (if the hash function is "random" enough)
  - On the attribute A: may be skewed (some values may appear more frequently than others)
- Range partition: may be skewed

Parallel Execution of RA Operators: Grouping

- Data: R(K,A,B,C)
- Query: select A, sum(C) from R group by R.A
- We consider the following situations
  - R is hash-partitioned on A
  - R is block-partitioned
  - R is hash-partitioned on K

Parallel Execution of RA Operators: Grouping

- Data: R(K,A,B,C)
- Query: select A, sum(C) from R group by R.A
- If R is block-partitioned or hash-partitioned on K

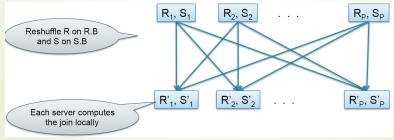


Speedup and Scaleup

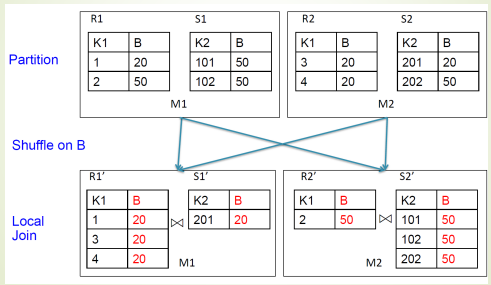
- Query: select A, sum(C) from R group by R.A
- Runtime: only consider IO/costs
  - Data distribution can be done in parallel
  - Suppose data is without skew
- We double the number of nodes
  - Running time becomes a half of the original
  - Each node holds only 50% number of chunks as before
- We double both the number of nodes and the size of R
  - Running time remains the same

Parallel Execution of RA Operators: Partitioned Hash-Join

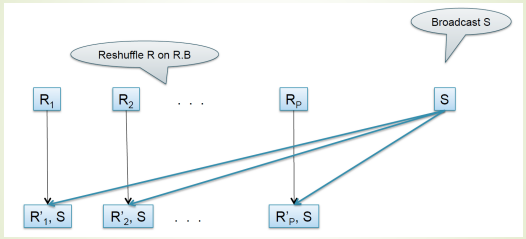
- Data: R(K1,A,B), S(K2,A,B)
- Query: R(K1,A,B)⋈S(K2,A,B)
  - Initially, both R and S are partitioned on K1 and K2



Parallel Join



Broadcast Join

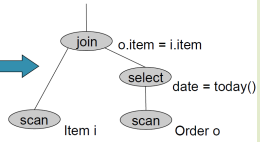


Why do we want to broadcast S?

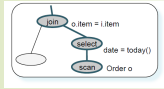
Putting it Together: Parallel Query Plan Example

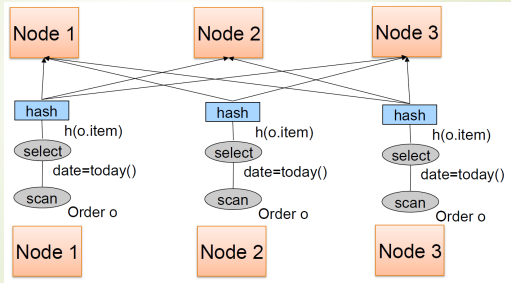
- Order(oid, item, date), Line(item, ...)
- We want to find all orders from today, along with the items ordered

```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```

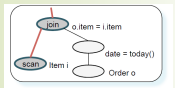


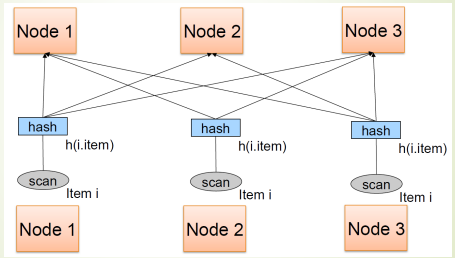
Parallel Query Plan Example



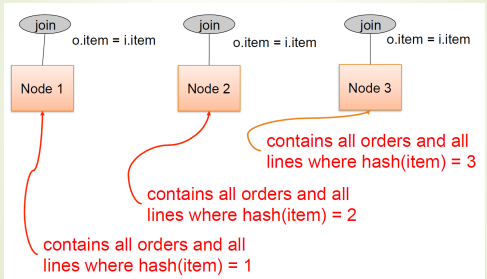


Parallel Query Plan Example





Parallel Query Plan Example



Why MapReduce?

Dealing with Large-scale Data

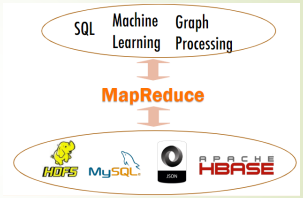
- Industry applications often involve large-scale datasets
  - Walmart: 2.5 petabytes of data every hour
  - How to store 1PB using 10,000 machines?
  - How to process 1PB using 10,000 machines?
- Solution: Distributed Computation Systems
- Challenges
  - How to distribute data?
  - Fault tolerant computing
  - How to make writing distributed programs easier?

Motivation

- We learned how to parallelize relational database systems
- It seems a lot of the distributed computation steps have similar pipelines (system thinking)
- MapReduce is a programming model for such computation

Why MapReduce?

- Fault Tolerant
  - Failure prob. 0.1%
  - 10,000 machines?
- Complex Analytics
- Heterogeneous Storage Systems



Hadoop/Spark

- Distributed Systems
  - Bring computation to data
  - Make copies of data for reliability
- Storage Infrastructure: Distributed File Systems
  - Google GFS, Hadoop HDFS
- Programming Model
  - MapReduce
  - Spark

Distributed File System

- **Chunk Servers/Nodes**
  - Files are split into chunks (with a fixed chunk size, normally 16-64MB)
  - Replicates for each chunk (2× or 3×)
  - Replicas are maintained in different chunk servers
- **Master Node/Name Node**
  - Stores metadata about where files are stored
  - May be replicated
- APIs for file access
  - Ask master node to find chunk servers
  - Connect to chunk servers to access data
- When some nodes fail, the system can recover data
  - Transparent to programmers (you do not need to worry about the recovery)

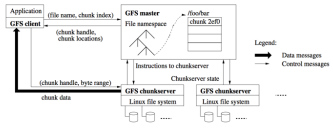
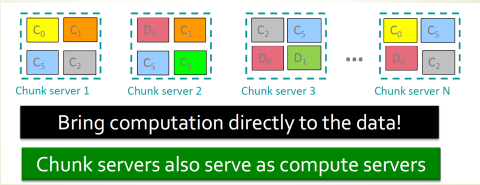


Figure 1: GFS Architecture

Chunks on different machines

- Replicates of chunks are spread across machines
  - If one machine fails, we can recover the data using other machines



MapReduce

MapReduce: the Programming Model

- How to distribute data?
  - What part of data should be put in what machines?
- MapReduce Model
  - Easy parallel programming
  - Management of hardware and software failures are transparent to programmers
  - Easy management of very large-scale datasets
- Many different implementations of MapReduce
  - Hadoop, Spark, Flink …

Typical Problems Solved by MR

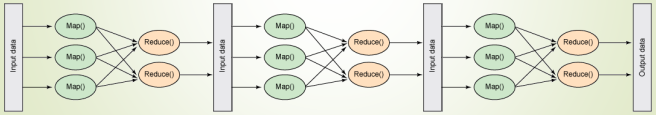
- Read a lot of data
- **Map**: extract something of interest from each record
- Shuffle and sort
- **Reduce**: aggregate, summarize, filter, transform the intermediate results
- Write the results

MapReduce Basics

- Foundational model is based on a distributed file system
  - Scalability and fault-tolerance
- **Map**
  - Apply a user-written *Map function* to each input element
  - The output of the Map function is a set of *key-value pairs*
- **Group by key**: Sort and shuffle
  - System sorts all the key-value pairs by key, and output *key-(list of values)* pairs
- **Reduce**
  - User-written *Reduce function* is applied to each key-(list of value) pair
- Performance can be tweaked based on known details of your source files and cluster shape (size, total number)

MapReduce Processing Model

- Define mappers
- Shuffling is automatic
- Define reducers
- For complex work, chain jobs together
  - Use a higher level language or DSL that does this for you



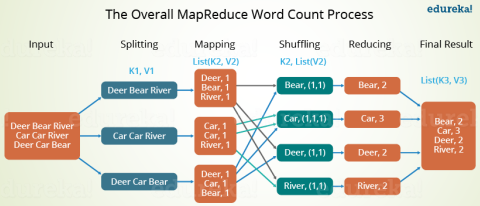
Vanilla MapReduce: Word Count

- Problem: Count the number of times each distinct word appears in the huge text files
- Applications
  - Analyze web server logs to find popular URLs
  - Statistical machine translation: Need to count number of times every 5-word sequence occurs in a large corpus of documents

```
map(key, value):
# key: document name; value: text of the document
  for each word w in value:
      emit(w, 1)


reduce(key, values):
# key: a word; value: an iterator over counts
      result = 0
      for each count v in values:
            result += v
      emit(key, result)
```

Word Count



Relational Operators in MapReduce

- Selection
- Projection
- Union
- Intersection
- Difference
- Join

Selection

- **Map process**
  - Input: (rowkey key, tuple g)
  - Check for each tuple, if selection conditions are satisfied.
  - Emit a pair (tuple g, null)
- **Reduce process**
  - Input: (tuple g, values)
  - For each input pair (tuple g, null), emit (tuple g, null)
  - Actually Reducer is not needed

Projection

- **Map process**
  - Input: (rowkey key, tuple g)
  - Tuple g -> Tuple t (by projection)
  - Emit(tuple t, null)
- **Reduce process**
  - Input: (tuple g, values)
  - Pairs with same keys are grouped together
  - Emit(tuple t, null)

Can reducer be skipped?

Union

- R∪S
- **Map process**
  - Input: (rowkey key, tuple g)
  - Emit(tuple g, null)
- **Reduce process**
  - Input: (tuple g, values)
  - Pairs with same keys are grouped together
  - Emit(tuple g, null)

Intersection

- R∩S
- **Map process**
  - Input: (rowkey key, tuple g)
  - ?
- **Reduce process**
  - Input: (tuple g, values)
  - ?

Intersection

- R∩S
- **Map process**
  - Input: (rowkey key, tuple g)
  - Emit a pair (tuple g, null)
- **Reduce process**
  - Input: (tuple g, values)
  - Emit(tuple g, null) if values.length=2

Difference

- R-S
- **Map process**
  - Input: (rowkey key, tuple g)
  - ?
- **Reduce process**
  - Input: (tuple g, values)
  - ?

Difference

- R-S
- **Map process**
  - Input: (rowkey key, tuple g)
  - Emit a pair (tuple g, setName) \\ R or S
- **Reduce process**
  - Input: (tuple g, values)
  - Emit(tuple g, null) if values.length=1 and values[0]=R

Join

- Compute the natural join $R(A,B) \bowtie S(B,C)$
- R and S are each stored in files
- Tuples are pairs (a,b) or (c,d)

A	B
a_1	b_1
a_2	b_2
a_3	b_3
a_4	b_4

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3
b_5	c_4

=

A	B	C
a_2	b_2	c_1
a_2	b_2	c_2
a_3	b_3	c_3

Join

- **Map process**
  - Input: (rowkey key, tuple g)
  - ?
- **Reduce process**
  - Input: (tuple g, values)
  - ?

Join

- **Map process**
  - Each input tuple $R(a,b)$, $\text{Emit}(b,(a,R))$
  - Each input tuple $S(b,c)$, $\text{Emit}(b,(c,S))$
- **Reduce process**
  - Matches all pairs $(b,(a,R))$ with all $(b,(c,S))$ and $\text{Emit}(b,(a,c))$

Behind the API

- Hadoop/Spark does the following jobs (transparent to users)
  - **Partitioning** the input (heterogeneous) data
  - **Scheduling** the program's execution across a set of machines
  - Performing the **group by key** step
  - Handling machine **failures**
  - Managing required inter-machine **communication**

MapReduce v.s. SQL

Map(k,v) ⟺ **SELECT** Map(v) **From** Table

Reduce(k,v_list) ⟺ **SELECT** Reduce(v) **From** Table **GROUP BY** k

MapReduce is a programming model rather than a DB system

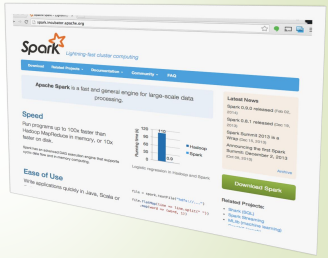
Spark

Problems with MapReduce

- Major limitations
  - Difficulty of programming directly in MapReduce
    - many problems aren't easily described as map-reduce
  - Performance bottlenecks, or batch not fitting the use cases
    - Persistence to disk typically slower than in-memory work
    - Not good for iterative computations (machine learning)
- **In short, MR doesn't compose well for large applications**
  - Many times one needs to chain multiple MapReduce steps

Spark: Data-Flow System

- Expressive computing system, not limited to MapReduce model
- Additions to MapReduce
  - Fast data sharing
    - In-memory computation
    - Caches data for repetitive queries (suitable for typical machine learning tasks)
  - General execution graphs
  - Richer functions (more than just map and reduce)
- Compatible with Hadoop



The Spark Community

Overview of Spark

- Open source
- Languages supported
  - Java, Scala and Python
- Key construct/idea: **Resilient Distributed Dataset (RDD)**
- Higher-level APIs: DataFrames & Datasets
  - SQL support
  - APIs for aggregating data

Resilient Distributed Dataset (RDD)

Resilient Distributed Dataset

- RDD = Resilient Distributed Datasets
  - A distributed, immutable relation, together with its *lineage*
  - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
  - If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD
- RDDs can be created from Hadoop, or by transforming other RDDs
- RDDs are best suited for applications that apply the same operation to all elements of a dataset, such as calculating gradients in machine learning tasks

Programming in Spark

- A Spark program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save…). Eager
- Eager: operators are executed immediately
- Lazy: operators are not executed immediately
  - An *operator tree* is constructed in memory instead
  - Similar to a relational algebra tree
- Collections in Spark
  - RDD<T> = an RDD collection of type T
    - Partitioned, recoverable (through lineage), not nested
  - Seq<T> = a sequence
    - Local to a server, may be nested

Example: Log Mining (in Java APIs)

- Given a large log file hdfs://logfile.log retrieve all lines that:
  - Start with "ERROR"

```
s = SparkSession.builder()...getOrCreate();

lines = s.read().textFile("hdfs://logfile.log");

errors = lines.filter(l -> l.startsWith("ERROR")); //l -> l.startsWith("ERROR") is anonymous functions (lambda expressions) starting in Java 8

errors.collect(); //JavaRDD<String> is the type of lines and errors
```

Example: Log Mining (in Java APIs)

- Given a large log file hdfs://logfile.log retrieve all lines that:
  - Start with "ERROR"

```
s = SparkSession.builder()...getOrCreate();

lines = s.read().textFile("hdfs://logfile.log");

errors = lines.filter(l -> l.startsWith("ERROR")); //l -> l.startsWith("ERROR") is anonymous functions (lambda expressions) starting in Java 8

errors.collect(); //JavaRDD<String> is the type of lines and errors
```

Lazy

Eager

MapReduce Again

Steps in Spark resemble MapReduce:

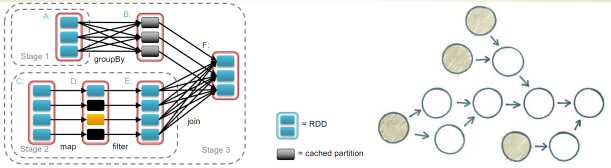
- col.filter(*p*) applies in parallel the predicate *p* to all elements *x* of the partitioned collection, and returns collection with those *x* where *p(x) = true*
- col.map(*f*) applies in parallel the function *f* to all elements *x* of the partitioned collection, and returns a new partitioned collection

Examples of Spark RDD Operations

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

Task Scheduler: General DAGs



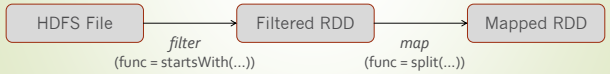
Spark supports general task graphs (DAGs)

- Nodes are tasks
- Edges indicate the flow
- Acyclic (No loops)
- Why important? Supports fault-tolerance

RDD Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```



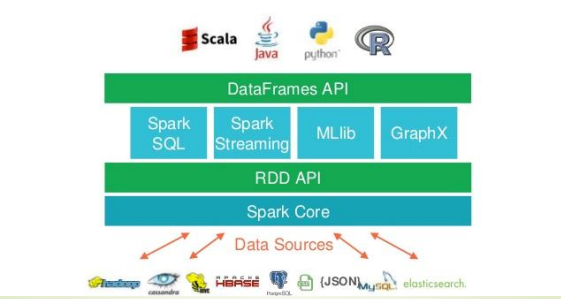
RDD Persistence / Caching

- Variety of storage levels
  - memory_only (default), memory_and_disk, etc…
- API Calls
  - persist(StorageLevel)
  - cache() – shorthand for persist(StorageLevel.MEMORY_ONLY)
- Considerations
  - Read from disk vs. recompute (memory_and_disk)
  - Total memory storage size (memory_only_ser)
  - Replicate to second node for faster fault recovery (memory_only_2)
    - Think about this option if supporting a web application

http://spark.apache.org/docs/latest/scala-programming-guide.html#rdd-persistence

Spark as a Platform

Unified Platform



Spark Streaming

- A scalable fault-tolerant streaming processing system
- Four major aspects
  - Fast recovery from failures and stragglers
  - Better load balancing and resource usage
  - Combining of streaming data with static datasets and interactive queries
  - Native integration with advanced processing libraries (SQL, machine learning, graph processing)

Machine Learning - MLlib

- K-Means
- L_1 and L_2-regularized Linear Regression
- L_1 and L_2-regularized Logistic Regression
- Alternating Least Squares
- Naive Bayes
- Stochastic Gradient Descent
- …

** Mahout is no longer accepting MapReduce algorithm submissions in lieu of Spark

Data Sources

- Local Files
  - file:///opt/httpd/logs/access_log
- S3
- Hadoop Distributed Filesystem
  - Regular files, sequence files, any other Hadoop InputFormat
- HBase

Deploying Spark – Cluster Manager Types

- Mesos
- EC2
- GCE
- Standalone mode
- YARN

Supported Languages

- Java
- Scala
- Python
- Hive

Spark SQL

DataFrames

- Like RDD, also an immutable distributed collection of data
- Organized into *named columns* rather than individual objects
  - Just like a relation
  - Elements are untyped objects called Row's
- Similar API as RDDs with additional methods

```
people = spark.read().textFile(…);
ageCol = people.col("age");
ageCol.plus(10); // creates a new DataFrame
```

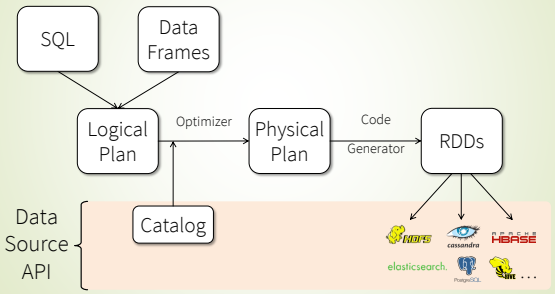
Datasets

- Similar to DataFrames, except that elements must be typed objects
  - E.g.: Dataset<People> rather than Dataset<Row>
- Can detect errors during compilation time
- DataFrames are aliased as Dataset<Row>

```
df.createOrReplaceTempView("people");
Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19|  Justin|
// +----+-------+
```

How to use Spark SQL: https://spark.apache.org/docs/2.2.0/sql-programming-guide.html

Execution Steps



Summary

- MapReduce
  - Why MapReduce
  - Examples of MapReduce
- Spark: distributed system based on MapReduce
  - RDD
  - Spark SQL

Readings

- **RG Chapter 22**
- MapReduce
  - https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html
- Spark Documentation
  - https://spark.apache.org/docs/latest/

Overview of Transaction Management

Instructor: Yu Yang
yuyang@cityu.edu.hk

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Motivation

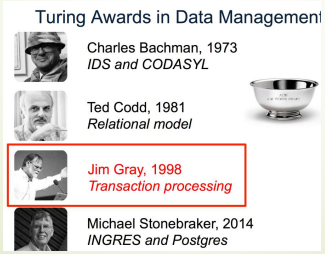
1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full, or not at all**; keep log to be able to "roll-back" TXNs

Why this lecture

- **A common issue in broad computing systems, not limited to DBMS**
  - What if crash occurs, power goes out, etc?
  - Single user → Multiple users



Transactions in SQL

- In "ad-hoc" SQL:
  - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
    UPDATE Bank SET amount = amount – 100
    WHERE name = 'Don'
    UPDATE Bank SET amount = amount + 100
    WHERE name = 'Joe'
COMMIT
```

Protection against crashes/aborts

```
Client 1:
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE Product
        WHERE price <=0.99
```

Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - Conflict Serializability

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.
2. **Concurrency:** Achieving better performance by parallelizing TXNs *without* creating anomalies

Protection against crashes/aborts

```
Client 1:
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE Product
        WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes/aborts

```
Client 1:
    START TRANSACTION
        INSERT INTO SmallProduct(name, price)
            SELECT pname, price
            FROM Product
            WHERE price <= 0.99

        DELETE Product
            WHERE price <=0.99
    COMMIT OR ROLLBACK
```

Now we'd be fine!

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming…

Multiple users: single statements

```
Client 1:   UPDATE Employee
            SET Salary = Salary + 1000

Client 2:   UPDATE Product
            SET Salary = Salary * 2
```

Two managers attempt to increase employee salary *concurrently*- What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
            UPDATE Employee
            SET Salary = Salary + 1000
          COMMIT

Client 2: START TRANSACTION
            UPDATE Employee
            SET Salary = Salary * 2
          COMMIT
```

Now works like a charm- we'll see how / why later…

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

ACID: Atomic

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Consistent

- The tables must always satisfy user-specified ***constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can't be negative
    - Sum of *debits* and of *credits* is 0
- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolated

- A transaction executes concurrently with other transactions
- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
  - E.g. Should not be able to observe changes from other transactions during the run

ACID: Durable

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…
- Means: Write data to **disk**

A Note: ACID is contentious!

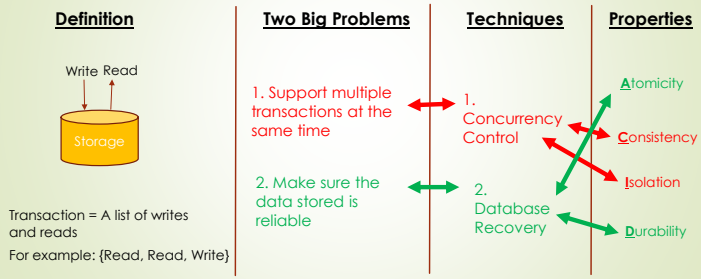
- Many debates over ACID, both **historically** and **currently**

- Many newer "NoSQL" DBMSs relax ACID

- In turn, now "NewSQL" reintroduces ACID compliance to NoSQL-style DBMSs…

ACID is an extremely important & successful paradigm, but still debated!

Transaction Management (Big Picture)



Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- **Concurrency Control**
  - **Scheduling**
  - **Anomaly types**
  - **Conflict serializability**

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…

1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
   - DBMS handles the details of *interleaving* various TXNs

   ACID

2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
   - DBMS handles the details of enforcing integrity constraints

   ACID

Example- consider two TXNs:

```
T1: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt + 100
      WHERE Name = 'A'

      UPDATE Accounts
      SET Amt = Amt - 100
      WHERE Name = 'B'
COMMIT
```

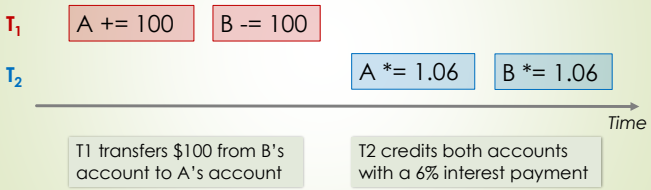
T1 transfers $100 from B's account to A's account

```
T2: START TRANSACTION
      UPDATE Accounts
      SET Amt = Amt * 1.06
COMMIT
```

T2 credits both accounts with a 6% interest payment

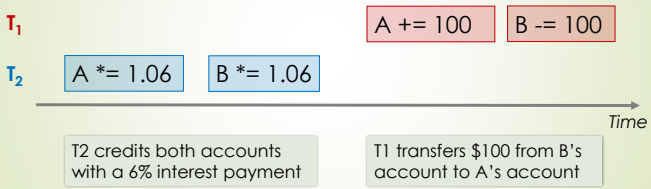
Example- consider two TXNs:

We can look at the TXNs in a timeline view- serial execution:



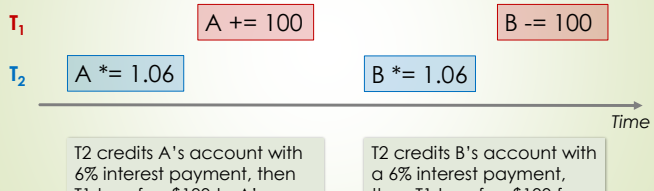
Example- consider two TXNs:

The TXNs could occur in either order… DBMS allows!



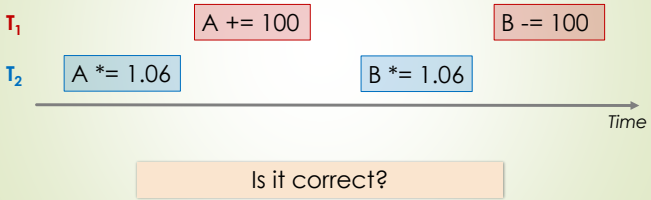
Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



Example- consider two TXNs:

The DBMS can also **interleave** the TXNs



Is it correct?

Why Interleave TXNs?

- Interleaving TXNs might lead to anomalous outcomes… why?
- Several important reasons:
  - Individual TXNs might be *slow*- don't want to block other users during!
  - Disk access may be *slow*- let some TXNs use CPUs while others accessing disk!

All concern large differences in ***performance***

Ignore all issues?

- At Facebook, only 0.0004% of results returned are inconsistent
- But,

Hacking, Distributed

NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex

nosql bitcoin mongo broken

Emin Gün Sirer — April 06, 2014 at 12:15 PM

← Older Newer →

Flexcoin was a Bitcoin exchange that shut down on March 3rd, 2014, when someone allegedly hacked in and made off with 896 BTC in the hot wallet.

Interleaving & Isolation

- The DBMS has freedom to interleave TXNs

"With great power comes great responsibility"

- However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained

ACID

  - Must be *as if* the TXNs had executed **serially**!

DBMS must pick a schedule which maintains isolation & consistency

Scheduling examples

Starting Balance: A $50, B $200

Serial schedule T_1,T_2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $159, B $106

Interleaved schedule 1:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $159, B $106

Same result!

Scheduling examples

Starting Balance: A $50, B $200

Serial schedule T_1,T_2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $159, B $106

Interleaved schedule 2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $159, B $112

Different result than serial T_1,T_2!

Scheduling examples

Starting Balance: A $50, B $200

Serial schedule T_2,T_1:

T_1: A+=100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $153, B $112

Interleaved schedule 2:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A $159, B $112

Different result than serial T_2,T_1 ALSO!

Scheduling examples

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

This schedule is different than ***any serial order!*** We say that it is **not serializable**

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions
- A **serializable schedule** is a schedule that is equivalent to ***some*** serial schedule.
- Schedule 1 and Schedule 2 are **equivalent** if, ***for any database state***, the effect on DB of executing Schedule 1 **is identical to** the effect of Schedule 2

Serializable?

Serial schedules:

	A	B
T_1,T_2	1.06*(A+100)	1.06*(B-100)
T_2,T_1	1.06*A + 100	1.06*B - 100

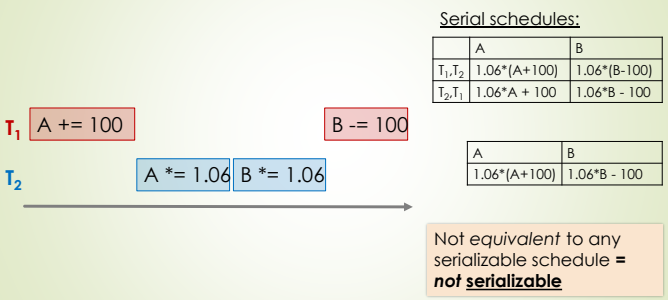
T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
1.06*(A+100)	1.06*(B-100)

Same as a serial schedule ***for all possible values of A, B*** = **serializable**

Serializable?



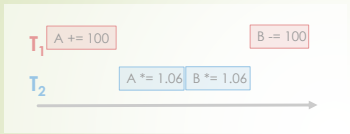
Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - **Anomaly types**
  - Conflict Serializability

What else can go wrong with interleaving?

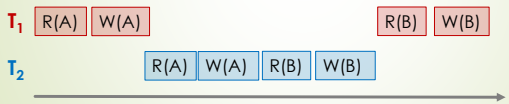
- Various anomalies which break isolation / serializability
  - Often referred to by name…
- Occur because of / with certain "conflicts" between interleaved TXNs

The DBMS's view of the schedule



Each action in the TXNs *reads a value from global memory* and then *writes one back to it*

Scheduling order matters!



Conflict Types

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

- Thus, there are three types of conflicts:
  - Read-Write conflicts (RW)
  - Write-Read conflicts (WR)
  - Write-Write conflicts (WW)

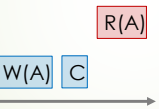
Why no "RR Conflict"?

Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

Classic Anomalies with Interleaved Execution

"Unrepeatable read":

Example:



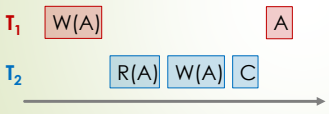
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again *and now gets a different / inconsistent value*

*Occurring with / because of a **RW conflict***

Classic Anomalies with Interleaved Execution

"Dirty read" / Reading uncommitted data:

Example:



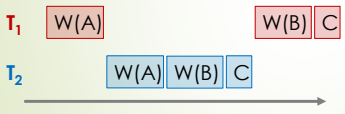
1. T_1 writes some data to A
2. T_2 reads from A, then writes back to A & commits
3. T_1 then aborts- *now T_2's result is based on an obsolete / inconsistent value*

*Occurring with / because of a **WR conflict***

Classic Anomalies with Interleaved Execution

"Lost update":

Example:



1. T_1 *blind writes* some data to A
2. T_2 *blind writes* to A *and B*
3. T_1 then *blind writes* to B; now we have T_2's value for B and T_1's value for A- ***not equivalent to any serial schedule!***

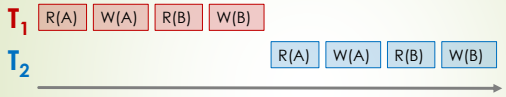
*Occurring because of a **WW conflict***

Outline

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - **Conflict Serializability**

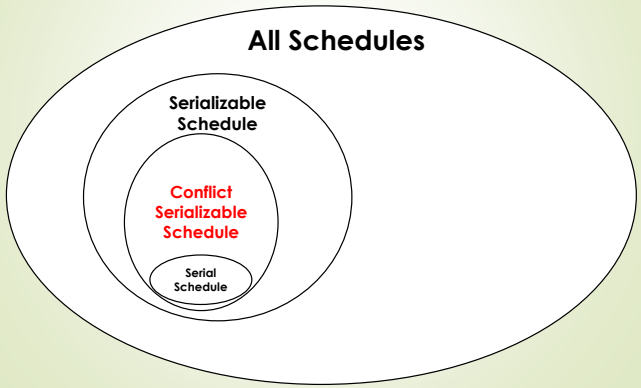
Schedules

Serial Schedule:



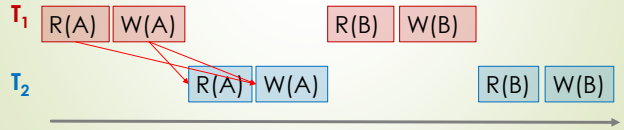
Serializable Schedule:

Conflict Serializable Schedule



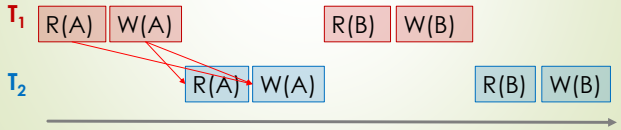
Conflicts

- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write



Exercise

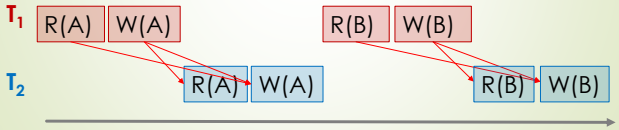
- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write



Find all the other conflicts

Exercise: Answer

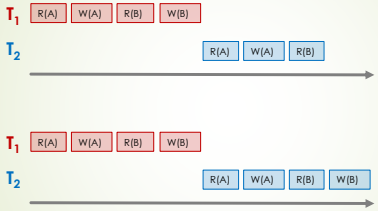
- Two actions **conflict** if all the conditions hold
  - i) they are part of different TXNs,
  - ii) they involve the same variable,
  - iii) at least one of them is a write



Conflict serializable

- Schedule S is **conflict serializable** if S is *conflict equivalent* to some serial schedule
- Two schedules are conflict equivalent if:
  - They involve *the same actions of the same TXNs*
  - Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

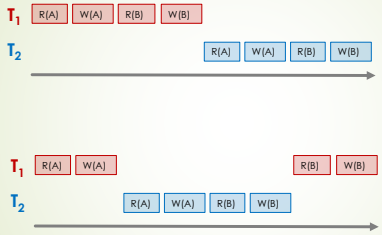
Are they conflict equivalent?



NO

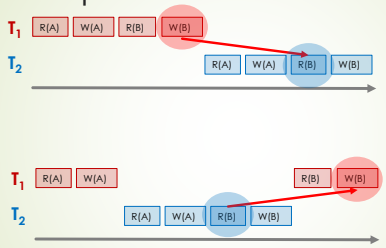
- "They involve *the same actions of the same TXNs*" does not hold

Are they conflict equivalent?



- "Every *pair of conflicting actions* of two TXNs are *ordered in the same way*" does not hold

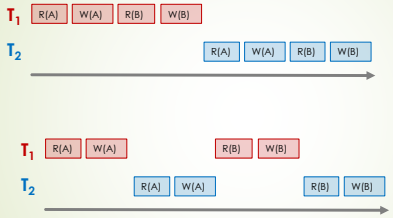
Are they conflict equivalent?



NO

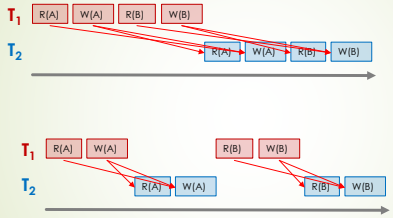
- "Every *pair of conflicting actions* of two TXNs are *ordered in the same way*" does not hold

Are they conflict equivalent?



- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

Are they conflict equivalent?



YES

- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

The Conflict Graph

- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i precede and conflict with any actions in T_j**

Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**

Is this schedule conflict serializable ?

1. Find all conflicts
2. Model the schedule as a conflict graph
3. Check whether the graph has a cycle
   - Yes → not conflict serializable
   - No → conflict serializable

NO

Isolation Levels

- Transactions in SQLite are serializable (https://www.sqlite.org/isolation.html)
- Isolation Levels in SQL Server (https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-2017)

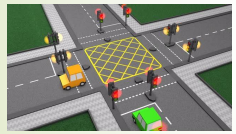
```
-- Syntax for SQL Server and Azure SQL Database

SET TRANSACTION ISOLATION LEVEL
    { READ UNCOMMITTED
    | READ COMMITTED
    | REPEATABLE READ
    | SNAPSHOT
    | SERIALIZABLE
    }
```

ACID

Concurrency Control Algorithms

- Locking



2-phase locking

- Timestamp Ordering

Multi-version concurrency control (MVCC)

Summary

- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties
- Concurrency Control
  - Scheduling
  - Anomaly Types
  - Conflict Serializability

Readings

- **RG Chapter 16, Chapter 17**
- GUW Chapter 18

SDSC5003 Course Review

Instructor: Yu Yang
yuyang@cityu.edu.hk

SDSC5003 Topics

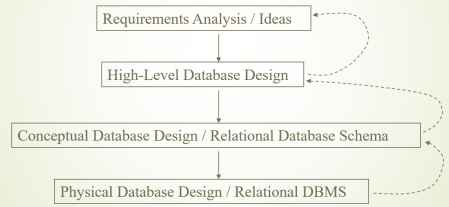
- Motivation: why storing and retrieving data?
- Data Modelling
  - ER Model
  - Relational Model
  - Principles of Database Design
- Data Manipulation
  - Relational Algebra
  - SQL: Constraints, triggers, applications ...
- Efficiency
  - Storage and Indexing
  - Query evaluation/optimization
- Parallelism
  - Parallel Data Processing
  - MapReduce/Spark
  - Transaction processing

Motivation

- Why we study data storing and retrieving?
  - Data is the ``fuel'' to data intensive applications
  - Frequent Routines: storing and retrieving (and updating) data
  - Big data
- One solution: Database Management System (DBMS)
  - Logical data independence
  - Physical data independence
  - Reduced application development time
  - Efficient access
  - Data integrity and security
  - Concurrent access and concurrency control
  - Crash recovery
  - ...

ER Model

- Database design



ER Model

- ER model basics
  - Entities
  - Relationships: binary, ternary, n-ary
  - Attributes
- Advanced ER model
  - Constraints: key, participation
  - Weak Entity
  - Aggregation
  - ...

Relational Model

- *Relational database:* a set of *relations*
- *Relation = Instance + Schema*
  - *Instance* : a *table*, with rows and columns.
    #Rows = *cardinality*, #fields = *degree or 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).

Relational Model

- Create relations/tables using SQL
  - Create
  - Drop
  - Alter
- Specifying constraints
  - Domain
  - Key
  - Foreign key
  - Other

```
CREATE TABLE Enrolled
  (sid CHAR(20),
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid,cid) )
CREATE TABLE Enrolled
  (sid CHAR(20)
   cid  CHAR(20),
   grade CHAR(2),
   PRIMARY KEY  (sid),
   UNIQUE (cid, grade) )
```

Relational Algebra

- Relational algebra is a procedural language that is used as the internal representation of SQL
  - The origins of SQL
  - Query processing and optimization
- Relational algebra basics
  - Relation instances (no duplicates)
    - Schema
  - Operations

Relational Algebra Operations

- Basic operations:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Selects a subset of 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 derived operations:
  - Intersection, *join*, division, renaming.
    Not essential, but very useful.
- Since each operation returns a relation, operations can be *composed*!

SQL

- SQL Syntax
  - *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 $<, >, =, \leq, \geq, \neq$) 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!

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

SQL Queries

- Set operations
  - Intersect, Union, Except
  - Join
- Nested queries
  - Correlated vs Non-correlated
- Aggregation
  - Count, Sum, AVG, …
  - Group By (Having)

```
SELECT B.brname, COUNT (A.accnum) AS accs
FROM Account A, Branch B
WHERE A.brname = B.brname AND
    B.budget > 500000
GROUP BY B.brname
HAVING SUM (A.balance) > 1000000
```

Design Theory

- Motivation: avoid anomaly
  - Redundancy

Design 1

Student	Course	Room
Mike	5003	LT-5
Mary	5003	LT-5
Sam	5003	LT-5
..	..	..

Design 2

Student	Course
Mike	5003
Mary	5003
Sam	5003
..	..

Course	Room
5003	LT-5
8005	7303

Design Theory

- Functional dependency

  A->B means that "whenever two tuples agree on A then they agree on B."
- Inference of FDs
- Normal forms
  - 1st, 2nd, **BCNF** (no bad FDs)
- BCNF decomposition
  - Lossless decoposition

Storage and Indexing

- Motivation: **Efficiency!**
  - Reducing I/Os
- File Organizations
  - 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 has some structures and can be organized 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.

Data Index

- Alternatives
  - Alternative 1: data records
  - Alternative 2, 3: data entries
- Clustered vs Unclustered
- Hash vs Tree
- Cost Analysis

Index-Only Plans

With a suitable index, some queries can be answered without retrieving any tuples from one or more relations involved

<E.dno,E.eid>
Tree index!

```
SELECT D.mgr, E.eid
FROM Dept D, Emp E
WHERE D.dno=E.dno
```

<E.dno>

```
SELECT E.dno, COUNT(*)
FROM Emp E
GROUP BY E.dno
```

<E.dno,E.sal>
Tree index!

```
SELECT E.dno, MIN(E.sal)
FROM Emp E
GROUP BY E.dno
```

<E. age,E.sal>
or
<E.sal, E.age>
Tree!

```
SELECT AVG(E.sal)
FROM Emp E
WHERE E.age=25 AND
 E.sal BETWEEN 3000 AND 5000
```

Query Evaluation & Optimization

- Motivation
  - The same query can be evaluated in different ways.
  - The evaluation strategy (plan) can make orders of magnitude of difference.
  - Query efficiency is one of the main areas where DBMS systems compete with each other.
  - Decades of development, secret details.

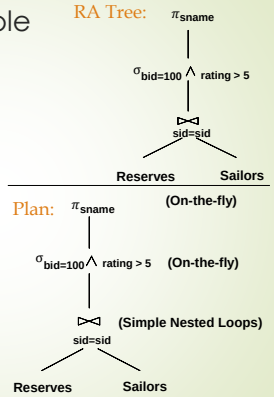
Query Plan

- ***Plan:*** *Tree of Relational Algebra operators, with choice of algorithms for each operator*
- Two main issues in query optimization:
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- Ideally: find best plan.
- Practically: Avoid worst plans!
- Query optimization
  - Cost model
  - Principle: execute more selective operations first

Query Plan Example

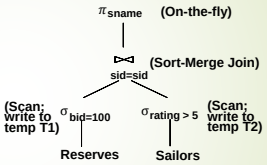
```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
  R.bid=100 AND S.rating>5
```

- RA Tree: expression tree.
- Each leaf is a schema table.
- Internal nodes: relational algebra operator applied to children.
- Full plan labels each internal node with implementation strategy.



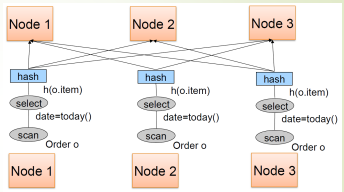
Alternative Plan

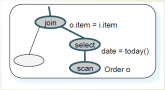
- *Goal of optimization:* To find efficient plans that compute the same answer.



Parallel Data Processing

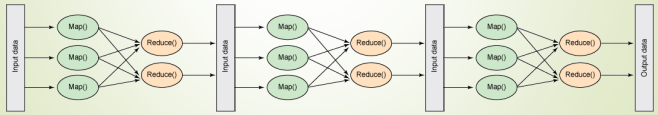
- Speedup and Scaleup
- Architectures
  - Shared Memory, Shared Disk, **Shared Nothing**
- Data partitioning
- Parallel Query Evaluation





Map-Reuce/Spark

- Distributed Computation Systems for Big Data
  - Fault tolerant
  - Complex analytics
  - Heterogeneous storage systems
- MapReduce: a programming model
- Spark system



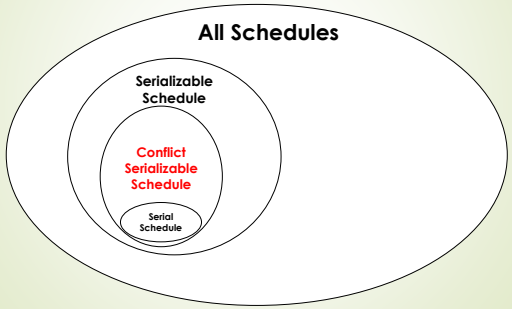
Transaction Processing

- Motivation
  - System crash
  - Multiple users
  - Concurrency
- Transaction Basics
  - Definition
  - Motivation for Transaction
  - ACID Properties

Transactions: Concurrency Control

- The DBMS must handle concurrency such that…
  1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**
     - DBMS handles the details of *interleaving* various TXNs
  2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**
     - DBMS handles the details of enforcing integrity constraints
- Anomaly types
- Schedules

Conflict Serializable Schedule



Final Exam

- DB Concepts (~5%)
- Relational Algebra & SQL (~35%)
- Design Theory (~10%)
- Storage & Indexing (~15%)
- Query Evaluation (~15%)
- Parallelism & MapReduce (~15%)
- Transaction Processing (~5%)

LOQ

Good luck for final exam!