# COSC 304
# Introduction to Database Systems

## Course Introduction

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## My Course Goals

My goals in teaching this course:

- Summarize and document the information in a simple, concise, and effective way for learning.
- Strive for ***all*** students to understand the material and pass the course.
- Be available for questions during class time, office hours, and at other times as needed.
- Provide a solid foundation on database systems and database application development.
- Teach students how to be a sophisticated database user (by understanding SQL), a database application programmer, and a database designer.
- Encourage students to continue with other database courses.





## Course Objectives

1) To learn the relational model and relational algebra

2) To be able to query databases using SQL

3) To understand how to build database applications

4) To practice designing databases using ER/UML diagrams

5) To develop with current database systems

6) To be exposed to XML/JSON and cloud databases





## Academic Dishonesty

Cheating in all its forms is strictly prohibited and will be taken very seriously by the instructor.

A guideline to what constitutes cheating:

- Assignments
  - ⇨ Working in groups comparing answers to questions once they have been solved.
  - ⇨ Copying code, even small code fragments, from other students.
  - ⇨ You may discuss general ideas and syntax, but never share code!
- Exams
  - ⇨ All exams are closed book, so no course materials should be present.

Cheating may result in a "F" for the assignment or course.





## Why this Course is Important

Database systems is a required course because most systems require a database as part of their implementation.

As a developer you will need to:

- ***Query/update databases*** – SQL and relational algebra are the languages to extract information from pre-existing databases.
- ***Program with databases*** – All languages require interfaces for retrieving/updating data from databases into program code for displaying reports or processing transactions.
- ***Design databases*** – Building a new application often involves storing data persistently in a database. The ability to design a database is a leading-edge skill.
- ***Be aware of data technologies*** – XML, NoSQL, JSON, web services, and a variety of other technologies are used for web and mobile applications.





## How to Pass This Course

The most important things to do to pass this course:

- Attend class
  - ⇨ Read notes *before* class.
- Do the assignments
  - ⇨ They are for marks, and they are good practice and exam questions.
- Develop a good project
  - ⇨ Spend time on selecting a good project that you will find interesting.
  - ⇨ Make sure to get started on the project early and budget sufficient time.
- Do additional questions
  - ⇨ Practice, practice, practice...

To get an “A” in this course do all the above plus:

- Do as many practice questions as possible.
- Create a really great project.

## The Database Implementation Project

A significant part of your mark for the course is devoted to a major database development project.

In this project you will:

- Design a database using ER/UML diagrams.
- Write SQL to query and update the database.
- Develop a web user interface to your database.
  - ⇨ In the process you will learn HTML and JSP/PHP.

These skills are highly valuable to potential employers. Note that limited background will be given on web programming.

The project can involve teams of 3 to 4. Larger teams must develop better projects.





## The Lab Assignments

Lab assignments are worth ***20%*** of your overall grade.

Lab assignments may take ***more than the two hours*** lab time.

You have at ***least one week*** after your lab to complete it.

- No late assignments will be accepted.
- An assignment may be handed in any time before the due date.

Lab assignments are done individually or in groups of two depending on the assignment.

***The lab assignments are critical to learning the material and are designed to prepare you for the exams!***

***Real-world experience with database development.***





## The In-Class Clicker Questions

To help with effort and understanding, ***5%*** of the overall grade is allocated to answering in-class questions using a clicker.

- The clicker can be purchased at the bookstore and sold back to the bookstore like a used textbook.
- The clicker is personalized to you with your student number.
- At different times during the lectures, questions reviewing material will be asked. Reponses are given using the clickers.

There will be at least 70 questions throughout the semester. Each question is worth 1 mark, and you need at least 50 right answers to get the full 5%.

- That is, if you answer 40 questions right, you get 40/50 or 4%.
- **No make-ups for forgetting clicker or missing class.**





## Systems and Tools

Connect is used for a discussion board, for posting marks, and for anonymous feedback.

- Please use the discussion board and feedback survey.

All software is available in the laboratory (SCI 234).

Access to both MySQL and SQL Server will be provided in the course. Access to a web server for web development will also be given. These systems will have separate user ids and passwords.





## My Expectations

My goal is for you to learn the material and walk out of this course confident in your abilities:

- To query and update an existing database
- Write code interfacing with a database to build standalone and web-based applications
- Be confident in your ability to design and model a database using UML

I have high standards on the amount and difficulty of material that we cover. I expect a strong, continual effort in keeping up with readings, doing assignments, and working on projects.

The course will be very straightforward – If you do the work, you will do well.

**Your mark is 80% perspiration and 20% inspiration.**





## Why are you here?

**A)** It is a required course for the COSC major.

**B)** This is an optional elective for my program.

## What Topic are You Most Interested In?

**A)** What is a database and how do you use them?

**B)** Querying using SQL

**C)** Designing databases

**D)** Using databases with programs (stand-alone, web, mobile)

**E)** None of the above





## What Grade are You Expecting to Get?

**A)** A

**B)** B

**C)** C

**D)** D

**E)** F





## Database Survey Question

***Question:*** Have you used any of these database systems?

**A)** MySQL

**B)** Microsoft Access or SQL Server

**C)** PostgreSQL

**D)** Used more than two different databases

**E)** Used no databases





## The Essence of the Course

If you walk out of this course with nothing else you should appreciate that:

Databases are the best way for storing and manipulating *persistent* information. You will learn the skills to exploit the full power of database systems.

The skills you will acquire are in high demand for many software development jobs. Database skills make you more marketable and allow you to construct more sophisticated systems.

- Note: This is a course on how to use/program with databases. It is a very applied course with specific skills.
- If you want to learn how to build database systems and what is "inside the box", that is the subject of COSC 404!

# COSC 304
# Introduction to Database Systems

## Database Introduction

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## What is a database?

A ***database*** is a collection of logically related data for a particular domain.

A ***database management system*** (***DBMS***) is software designed for the creation and management of databases.

- e.g. Oracle, DB2, Access, MySQL, SQL Server, MongoDB

Bottom line: A ***database*** is the *data* stored and a ***database system*** is the *software* that manages the data.





## Databases in the Real-World

Databases are everywhere in the real-world even though you do not often interact with them directly.

- $25 billion dollar annual industry

Examples:

- Retailers manage their products and sales using a database.
  - ⇨Wal-Mart has one of the largest databases in the world!
- Online web sites such as Amazon, eBay, and Expedia track orders, shipments, and customers using databases.
- The university maintains all your registration information and marks in a database.

Can you think of other examples?

What data do ***you*** have?





## Example Problem

Implement a system for managing products for a retailer.

- Data: Information on products (SKU, name, desc, inventory)
- Add new products, manage inventory of products

How would you do this without a database?

What types of challenges would you face?





## Why do we need databases?

Without a DBMS, your application must rely on files to store its data *persistently*. A ***file-based system*** is a set of applications that use files to store their data.

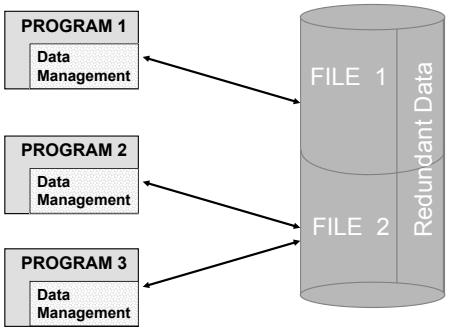
Each application in a file-based system contains its own code for accessing and manipulating files. This causes several problems:

- Code duplication of file access routines
- Data is usually highly redundant across files
- High maintenance costs
- Hard to support multi-user access to information
- Difficult to connect information present in different files
- Difficulty in developing new applications/handling data changes





## File Processing Diagram



## Data Independence and Abstraction

The major problem with developing applications based on files is that the application is dependent on the file structure.

That is, there is no ***program-data independence*** separating the application from the data it is manipulating.

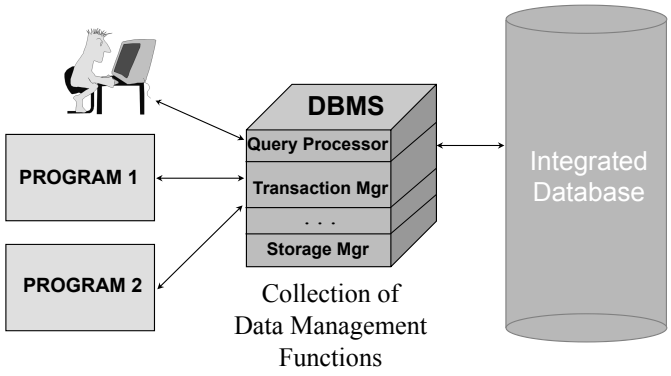
- If the data file changes, the code that accesses the file must be changed in the application.

One of the major advantages of databases is they provide data abstraction. ***Data abstraction*** allows the internal definition of an object to change without affecting programs that use the object through an external definition.





## Database System Approach








## DBMS

A database management system provides *efficient*, *convenient*, and *safe multi-user* storage and access to *massive* amounts of *persistent* data.

***Efficient*** - Able to handle large data sets and complex queries without searching all files and data items.

***Convenient*** - Easy to write queries to retrieve data.

***Safe*** - Protects data from system failures and hackers.

***Massive*** - Database sizes in gigabytes/terabytes/petabytes.

***Persistent*** - Data exists after program execution completes.

***Multi-user*** - More than one user can access and update data at the same time while preserving consistency.





## Data Definition Language

A DBMS achieves these goals by supporting data abstraction.

- The DBMS takes the description of the data and handles the low-level details of how to store it, retrieve it, and handle concurrent access to it.

The database is described to the DBMS using a ***Data Definition Language*** (***DDL***). The DDL allows the user to create data structures in the data model used by the database.

A ***data model*** is a collection of concepts that can be used to describe the structure of a database.

- In the relational model, data is represented as tables and fields.
- Examples: relational model, XML, graphs, object-oriented, JSON





## Schemas

A database designer uses a DDL to define a schema for the database. The schema is maintained and stored in the ***system catalog***. The schema is one type of ***metadata***.

A ***schema*** is a description of the structure of the database.

- A schema contains structures, names, and types of data stored.
- For example, the data model for Access is a relational model. A relational model contains tables and fields as its model constructs. The following DDL creates a product table:

```
CREATE TABLE product(
    sku as VARCHAR(10) NOT NULL,
    name as VARCHAR(40),
    desc as VARCHAR(50),
    inventory as INTEGER,
    PRIMARY KEY (sku)
);
```





## Data Manipulation Language

Once a database has been created using DDL, the user accesses data using a ***Data Manipulation Language*** (***DML***).

- The standard DML is SQL.
- The DML allows for the insertion, modification, retrieval, and deletion of data.

A DML provides data abstraction as user queries are specified using the names of data and not their physical representation.

- For example, in a file system storing 3 fields, you would have to provide the exact location of the field in the file. In a database, you would only have to specify it by name.
- The DBMS contains all the code for accessing the data, so the applications do not have to worry about those details any more.

## SQL Examples

Retrieve all products in the database:

```
SELECT sku, name, desc, inventory FROM product;
```

Retrieve all products where inventory < 10:

```
SELECT name, inventory FROM product WHERE inventory < 10;
```

Insert a new product into the database:

```
INSERT INTO product VALUES ('1234','Soap','Ivory',100);
```

Delete a product from the database:

```
DELETE FROM product WHERE sku = '1234';
```





## Database Properties Question

***Question:*** **True or False**: The data in a database is lost when the power to the computer is turned off.

**A)** true

**B)** false





## Database Abstraction Question

***Question:*** Defining how data is stored using DDL is similar to what in object-oriented programming?

**A)** Objects

**B)** Classes

**C)** Inheritance

**D)** Polymorphism





## DDL vs. DML Question

***Question:*** If you are querying data in a database, which language are you using:

**A)** DML

**B)** DDL

**C)** schemas

**D)** Java





## Components of a DBMS

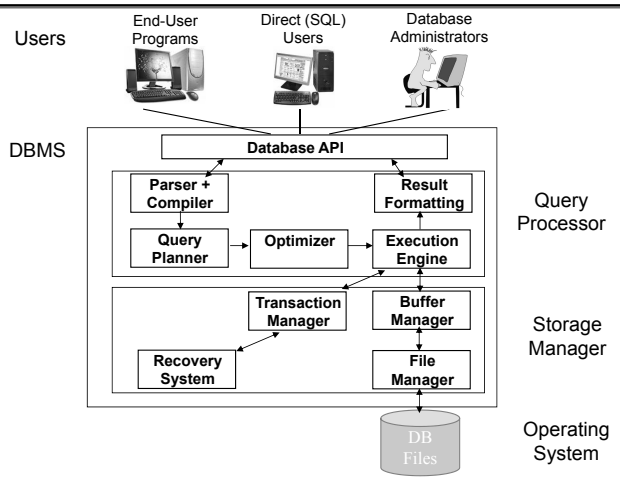
A DBMS is a complicated software system containing many components:

- ***Query processor*** - translates user/application queries into low-level data manipulation actions.
  - ⇨Sub-components: query parser, query optimizer
- ***Storage manager*** - maintains storage information including memory allocation, buffer management, and file storage.
  - ⇨Sub-components: buffer manager, file manager
- ***Transaction manager*** - performs scheduling of operations and implements concurrency control algorithms.





## DBMS Architecture



## Database Architectures

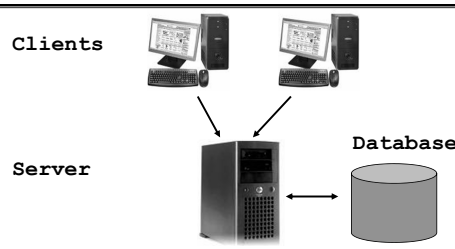
There are several different database architectures:

- **File-server (embedded) architecture** - files are shared but DBMS processing occurs at the clients (e.g. Microsoft Access or SQLite)
- **Two-Tier client-server architecture** - dedicated machine running DBMS accessed by clients (e.g. SQL Server)
- **Three-Tier client-server architecture** - DBMS is bottom tier, second tier is an application server containing business logic, top tier is clients (e.g. Web browser-Apache/Tomcat-Oracle)





## Two-Tier Client-Server Architecture




Tier 1: Client
•User Interface
•Business and Data Logic

Tier 2: Database Server
•Server-side validation
•Data storage/management

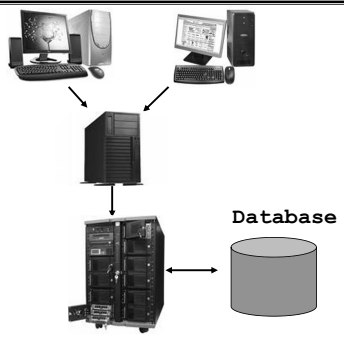
Advantages:

- Only one copy of DBMS software on dedicated machine.
- Increased performance.
- Reduced hardware and communication costs.
- Easier to maintain consistency and manage concurrency.





## Three-Tier Client-Server Architecture




Tier 1: Client (Web/mobile)
•User Interface

Tier 2: Application Server
•Business logic
•Data processing logic

Tier 3: Database Server
•Data validation
•Data storage/management

Advantages:

- Reduced client administration and cost using thin web clients.
- Easy to scale architecture and perform load balancing.





## Database People

There are several different types of database personnel:

- ***Database administrator*** (***DBA***) - responsible for installing, maintaining, and configuring the DBMS software.
- ***Data administrator*** (***DA***) - responsible for organizational policies on data creation, security, and planning.
- ***Database designer*** - defines and implements a schema for a database and associated applications.
  - ⇨***Logical/Conceptual database designer*** - interacts with users to determine data requirements, constraints, and business rules.
  - ⇨***Physical database designer*** - implements the logical design for a data model on a DBMS. Defines indexes, security, and constraints.
- ***DBMS developer*** - writes the DBMS software code.
- ***Application developer*** - writes code that uses the DBMS.
- ***User*** - uses the database directly or through applications.





## ANSI/SPARC Architecture

One of the major advantages of database systems is data abstraction. Data abstraction is achieved by defining different views of the data. Each view isolates higher-level views from data representation details.

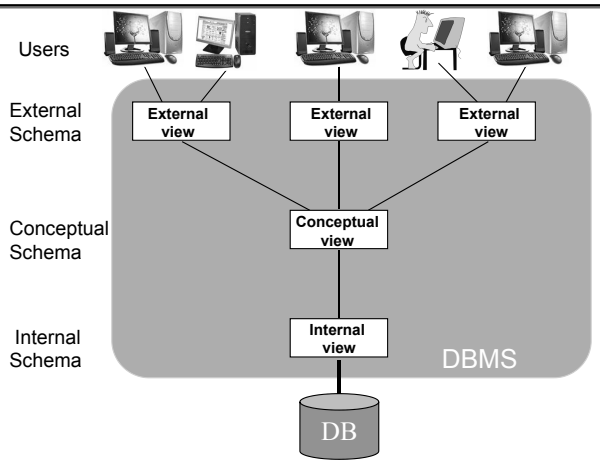
The ANSI/SPARC architecture defined in 1975 consists of three views:

- ***Internal View*** - The physical representation of the database on the computer. *How* the data is stored.
- ***Conceptual View*** - The logical structure of the database that describes *what* data is stored and its relationships.
- ***External View*** - The *user's view* of the database that provides the part of the database relevant to the user.





## ANSI/SPARC Architecture




3-schema architecture

## Benefits of 3-Schema Architecture

External Level:

- Each user can access the data, but have their own view of the data independent of other users.
  - ⇨ *Logical data independence* - conceptual schema changes do not affect external views.

Conceptual Level:

- Single shared data representation for all applications and users which is independent of physical data storage.
  - ⇨ Users do not have to understand physical data representation details.
  - ⇨ The DBA can change the storage structures without affecting users or applications. *Physical data independence* - conceptual schema not affected by physical changes such as adding indexes or distributing data.

Internal (Physical) Level:

- Provides standard facilities for interacting with operating system for space allocation and file manipulation.





## Microsoft Access and the 3-Schema Architecture

External Level:

- Microsoft Access does not call them views, but you can store queries and use the results in other queries (like a view).
  - ⇨ External schema is the query (view) name and the attribute metadata.

Conceptual Level:

- All tables and field definitions are in the schema (accessible from the Tables tab).
  - ⇨ Note that conceptual **schema** is not the data but the metadata.

Physical Level:

- Access represents all data in a single file whose layout it controls.
- The system processes this raw data file by knowing locations and offsets of relations and fields.





## ANSI/SPARC Architecture Three Levels of Views

***Question:*** What are the three levels of views in the ANSI/SPARC architecture starting with the view closest to the user?

**A)** Physical, Conceptual, External

**B)** External, Physical, Conceptual

**C)** Physical, External, Conceptual

**D)** External, Conceptual, Physical

**E)** User, Logical, System





## ANSI/SPARC Architecture Abstraction with Views

***Question:*** Assume you have a Java program accessing data stored in a file. Select ***one*** true statement.

**A)** The file organization is changed. The physical view is where this change is made.

**B)** A field is added to the database. The conceptual view is changed.

**C)** A user account has restricted access to the file. The external view must be changed.

**D)** More than one of the above





## Conclusion

A database is a collection of logically related data stored and managed by a database management system (DBMS).

A DBMS has advantages over traditional file systems as they support data independence and provide standard implementations for data management tasks.

- Data definition and manipulation languages (DDL and DML)
- System catalog maintains database description (schema) defined using the data model.

The 3-schema architecture consists of external, conceptual, and physical schemas. Each view provides data abstraction and isolates the layer above from certain data manipulation details.





## Objectives

- Define: database, DBMS, database application/system
- Describe the features of a file-based system and some limitations inherent in that architecture.
- Define program-data independence and explain how it is achieved by databases but not by file systems.
- Define DDL and DML. What is the difference?
- List some modules of a DBMS.
- List different people associated with a DBMS and their roles.
- Explain how a schema differs from data.
- Draw a diagram of the 3-schema architecture and explain what each level provides. List benefits of the architecture.
- How does a schema provide data independence?
- Compare/contrast two-tier and three-tier architectures.

# COSC 304
# Introduction to Database Systems

## Relational Model and Algebra

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Relational Model History

The relational model was proposed by E. F. Codd in 1970.

One of the first relational database systems, System R, developed at IBM led to several important breakthroughs:

- the first version of SQL
- various commercial products such as Oracle and DB2
- extensive research on concurrency control, transaction management, and query processing and optimization

Commercial implementations (RDBMSs) appeared in the late 1970s and early 1980s. Currently, the relational model is the foundation of the majority of commercial database systems.





## Relational Model Definitions

A ***relation*** is a table with columns and rows.

An ***attribute*** is a named column of a relation.

A ***tuple*** is a row of a relation.

A ***domain*** is a set of allowable values for one or more attributes.

The ***degree*** of a relation is the number of attributes it contains.

The ***cardinality*** of a relation is the number of tuples it contains.

A ***relational database*** is a collection of normalized relations with distinct relation names.

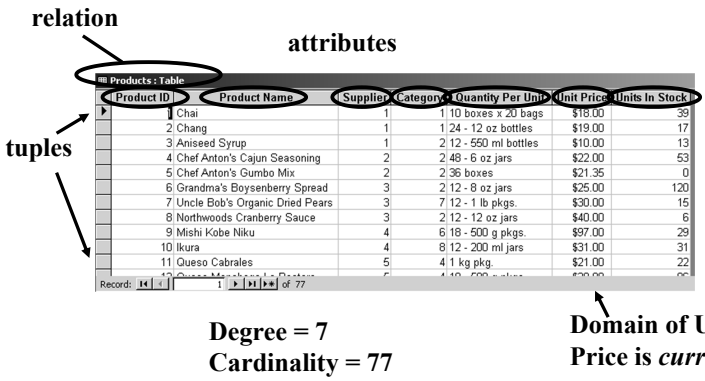
The ***intension*** of a relation is the structure of the relation including its domains.

The ***extension*** of a relation is the set of tuples currently in the relation.





## Relation Example








## Definition Matching Question

***Question:*** Given the three definitions, select the ordering that contains their related definitions.

**1)** relation
**2)** tuple
**3)** attribute

**A)** column, row, table
**B)** row, column, table
**C)** table, row, column
**D)** table, column, row





## Cardinality and Degree Question

***Question:*** A database table has 10 rows and 5 columns. Select ***one*** true statement.

**A)** The table's degree is 50.

**B)** The table's cardinality is 5.

**C)** The table's degree is 10.

**D)** The table's cardinality is 10.

# Relation Practice Questions

Order : Select Query

| Order ID | Customer | Employee | Order Date | Shipped Date | Ship Via | Ship Name | Ship Address | Ship Postal Code |
|---|---|---|---|---|---|---|---|---|
| 10248 | VINET | 5 | 04-Aug-94 | 16-Aug-94 | 3 | Vins et alcools Chevalier | 59 rue de l'Abbaye | 51100 |
| 10249 | TOMSP | 6 | 05-Aug-94 | 10-Aug-94 | 1 | Toms Spezialitäten | Luisenstr. 48 | 44087 |
| 10250 | HANAR | 4 | 08-Aug-94 | 12-Aug-94 | 2 | Hanari Carnes | Rua do Paço, 67 | 05454-876 |
| 10251 | VICTE | 3 | 08-Aug-94 | 15-Aug-94 | 1 | Victuailles en stock | 2, rue du Commerce | 69004 |
| 10252 | SUPRD | 4 | 09-Aug-94 | 11-Aug-94 | 2 | Suprêmes délices | Boulevard Tirou, 255 | B-6000 |
| 10253 | HANAR | 3 | 10-Aug-94 | 16-Aug-94 | 2 | Hanari Carnes | Rua do Paço, 67 | 05454-876 |
| 10254 | CHOPS | 5 | 11-Aug-94 | 23-Aug-94 | 2 | Chop-suey Chinese | Hauptstr. 31 | 3012 |
| 10255 | RICSU | 9 | 12-Aug-94 | 15-Aug-94 | 3 | Richter Supermarkt | Starenweg 5 | 1204 |
| 10256 | WELLI | 3 | 15-Aug-94 | 17-Aug-94 | 2 | Wellington Importadora | Rua do Mercado, 12 | 08737-363 |
| 10257 | HILAA | 4 | 16-Aug-94 | 22-Aug-94 | 3 | HILARIÓN-Abastos | Carrera 22 con Ave. Carlos | 5022 |
| 10258 | ERNSH | 1 | 17-Aug-94 | 23-Aug-94 | 1 | Ernst Handel | Kirchgasse 6 | 8010 |
| 10259 | CENTC | 4 | 18-Aug-94 | 25-Aug-94 | 3 | Centro comercial Moctezuma | Sierras de Granada 9993 | 05022 |
| 10260 | OTTIK | 4 | 19-Aug-94 | 29-Aug-94 | 1 | Ottilies Käseladen | Mehrheimerstr. 369 | 50739 |

Record: 1 of 827

1) What is the name of the relation?

2) What is the cardinality of the relation?

3) What is the degree of the relation?

4) What is the domain of order date? What is the domain of order id?

5) What is larger the size of the intension or extension?





# Relational Model Formal Definition

The relational model may be visualized as tables and fields, but it is formally defined in terms of sets and set operations.

A ***relation schema R*** with attributes $\boldsymbol{A} = <A_1, A_2, \ldots, A_n>$ is denoted $\boldsymbol{R}(A_1, A_2, \ldots, A_n)$ where each $A_i$ is an attribute name that ranges over a domain $D_i$ denoted $\text{dom}(A_i)$.

Example: Product (id, name, supplierId, categoryId, price)

- ***R*** = Product (relation name)
- Set ***A*** = {id, name, supplierId, categoryId, price}
- dom(price) is set of all possible positive currency values
- dom(name) is set of all possible strings that represent people's names





# Relation Schemas and Instances

A ***relation schema*** is a definition of a single relation.

- The relation schema is the *intension* of the relation.

A ***relational database schema*** is a set of relation schemas (modeling a particular domain).

A ***relation instance*** denoted $r(R)$ over a relation schema $\boldsymbol{R}(A_1, A_2, \ldots, A_n)$ is a set of n-tuples $<d_1, d_2, ..., d_n>$ where each $d_i$ is an element of $\text{dom}(A_i)$ or is **null**.

- The relation instance is the *extension* of the relation.
- A value of **null** represents a missing or unknown value.





# Cartesian Product (review)

The ***Cartesian product*** written as $D_1 \times D_2$ is a set operation that takes two sets $D_1$ and $D_2$ and returns the set of all ordered pairs such that the first element is a member of $D_1$ and the second element is a member of $D_2$.

Example:

- $D_1 = \{1,2,3\}$
- $D_2 = \{A,B\}$
- $D_1 \times D_2 = \{(1,A), (2,A), (3,A), (1,B), (2,B), (3,B)\}$

Practice Questions:

- 1) Compute $D_2 \times D_1$.
- 2) Compute $D_2 \times D_2$.
- 3) If $|D|$ denotes the number of elements in set $D$, how many elements are there in $D_1 \times D_2$ in general.
  - ⇨What is the *cardinality* of $D_1 \times D_2 \times D_1 \times D_1$? A) 27 B) 36 C) 54





# Relation Instance

A ***relation instance*** **r(R)** can also be defined as a subset of the *Cartesian product* of the *domains* of all *attributes* in the *relation schema*. That is,

$$\mathbf{r(R)} \subseteq \text{dom}(A_1) \times \text{dom}(A_2) \times \ldots \times \text{dom}(A_n)$$

Example:

- R = Person(id, firstName, lastName)
- dom(id) = {1,2}, dom(firstName) = {Joe, Steve}
- dom(lastName) = {Jones, Perry}
- dom(id) × dom(firstName) × dom(lastName) =
  - ⇨{ (1,Joe,Jones), (1,Joe,Perry), (1,Steve,Jones), (1,Steve,Perry), (2,Joe,Jones), (2,Joe,Perry), (2,Steve,Jones), (2,Steve,Perry)}
- Assume our DB stores people Joe Jones and Steve Perry, then r(R) = { (1,Joe, Jones), (2,Steve,Perry)}.





# Properties of Relations

A relation has several properties:

- 1) Each relation name is unique.
  - ⇨No two relations have the same name.
- 2) Each cell of the relation (value of a domain) contains exactly one atomic (single) value.
- 3) Each attribute of a relation has a distinct name.
- 4) The values of an attribute are all from the same domain.
- 5) Each tuple is distinct. There are no duplicate tuples.
  - ⇨This is because relations are sets. In SQL, relations are bags.
- 6) The order of attributes is not really important.
  - ⇨Note that this is different that a mathematical relation and our definitions which specify an ordered tuple. The reason is that the attribute names represent the domain and can be reordered.
- 7) The order of tuples has no significance.

# Relational Keys

Keys are used to uniquely identify a tuple in a relation.

- ⇨Note that keys apply to the relational schema not to the relational instance. That is, looking at the current data cannot tell you for sure if the set of attributes is a key.

A ***superkey*** is a set of attributes that uniquely identifies a tuple in a relation.

A ***key*** is a *minimal* set of attributes that uniquely identifies a tuple in a relation.

A ***candidate key*** is one of the possible keys of a relation.

A ***primary key*** is the candidate key designated as the distinguishing key of a relation.

A ***foreign key*** is a set of attributes in one relation referring to the primary key of another relation.

- ⇨Foreign keys allow referential integrity to be enforced.





# Keys and Superkeys Question

***Question:*** **True or false:** A key is always a superkey.

**A)** true

**B)** false





# Keys and Superkeys Question (2)

***Question:*** **True or false:** It is possible to have more than one key for a table and the keys may have different numbers of attributes.

**A)** true

**B)** false





# Keys and Superkeys Question (3)

***Question:*** **True or false:** It is possible to always determine if a field is a key by looking at the data in the table.

**A)** true

**B)** false





# Example Relations

Employee-Project Database:

- ◆Employees have a unique number, name, title, and salary.
- ◆Projects have a unique number, name, and budget.
- ◆An employee may work on multiple projects and a project may have multiple employees. An employee on a project has a particular responsibility and duration on the project.

Relations:

Emp (<u>eno</u>, ename, title, salary)

Proj (<u>pno</u>, pname, budget)

WorksOn (<u>eno</u>, <u>pno</u>, resp, dur)

Underlined attributes denote keys.





# Example Relation Instances

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

Questions:
1) Is *ename* a key for *emp*?
2) Is *eno* a key for *WorksOn*?
3) List all the superkeys for *WorksOn*.

# Practice Questions

Consider a relation storing driver information including:

- SSN, name, driver's license number and state (unique together)

Person Relation

| SSN | name | LicNum | LicState |
|---|---|---|---|
| 123-45-6789 | S. Smith | 123-456 | IA |
| 111-11-1111 | A. Lee | 123-456 | NY |
| 222-22-2222 | J. Miller | 555-111 | MT |
| 333-33-3333 | B. Casey | 678-123 | OH |
| 444-44-4444 | A. Adler | 456-345 | IA |

Questions:
1) List the candidate keys for the relation.
2) Pick a primary key for the relation.
3) Is *name* a candidate key for *Person*?
4) List all the superkeys for *Person*.

Assumptions:
1) A person has only one driver's license.
2) A driver's license uniquely identifies a person.





# Relational Integrity

Integrity rules are used to insure the data is accurate.

***Constraints*** are rules or restrictions that apply to the database and limit the data values it may store.

Types of constraints:

- ***Domain constraint*** - Every value for an attribute must be an element of the attribute's domain or be `null`.
  - ⇨`null` represents a value that is currently unknown or not applicable.
  - ⇨`null` is not the same as zero or an empty string.
- ***Entity integrity constraint*** - In a base relation, no attribute of a primary key can be null.
- ***Referential integrity constraint*** - If a foreign key exists in a relation, then the foreign key value must match a primary key value of a tuple in the referenced relation or be null.





# Foreign Keys Example

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

**WorksOn.eno is FK to Emp.eno**

**WorksOn.pno is FK to Proj.pno**

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |





# Foreign Keys Example (2)

Proj Relation

| pno | pname | budget | dno |
|---|---|---|---|
| P1 | Instruments | 150000 | D1 |
| P2 | DB Develop | 135000 | D2 |
| P3 | CAD/CAM | 250000 | D3 |
| P4 | Maintenance | 310000 | null |
| P5 | CAD/CAM | 500000 | D1 |

**Proj.dno is FK to Dept.dno**

Department Relation

| dno | dname |
|---|---|
| D1 | Management |
| D2 | Consulting |
| D3 | Accounting |
| D4 | Development |





# Integrity Constraints Question

***Question:*** What constraint says that a primary key field cannot be null?

**A)** domain constraint

**B)** referential integrity constraint

**C)** entity integrity constraint





# Entity Integrity Constraint Question

***Question:*** A primary key has three fields. Only one field is `null`. Is the entity integrity constraint violated?

**A)** Yes

**B)** No

## Referential Integrity Constraint Question

***Question:*** A foreign key has a `null` value in the table that contains the foreign key fields. Is the referential integrity constraint violated?

**A)** Yes

**B)** No





## Integrity Questions

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | AS |
| E2 | null | SA | 50000 |
| E3 | A. Lee | 12 | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| null | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | null |
| E8 | J. Jones | SA | 50000 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | null | null |

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P0 | null | 12 |
| E2 | P1 | Analyst | null |
| null | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E9 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | null | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P6 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |
| null | null | Manager | 40 |

Question:
How many violations of integrity constraints?
A) 8 B) 9 C) 10 D) 11 E) 12





## General Constraints

There are more general constraints that some DBMSs can enforce. These constraints are often called ***enterprise constraints*** or ***semantic integrity constraints***.

Examples:

- An employee cannot work on more than 2 projects.
- An employee cannot make more money than their manager.
- An employee must be assigned to at least one project.

Ensuring the database follows these constraints is usually achieved using triggers.





## Relational Algebra

A ***query language*** is used to update and retrieve data that is stored in a data model.

***Relational algebra*** is a set of relational operations for retrieving data.

- Just like algebra with numbers, relational algebra consists of operands (which are relations) and a set of operators.

Every relational operator takes as input one or more relations and produces a relation as output.

- Closure property - input is relations, output is relations
- Unary operations - operate on one relation
- Binary operations - have two relations as input

A sequence of relational algebra operators is called a ***relational algebra expression***.





## Relational Algebra Operators

Relational Operators:

- Selection $\sigma$
- Projection $\Pi$
- Cartesian product $\times$
- Join $\bowtie$
- Union $\cup$
- Difference -
- Intersection $\cap$

Note that relational algebra is the foundation of ALL relational database systems. SQL gets translated into relational algebra.





## Selection Operation

The ***selection operation*** is a unary operation that takes in a relation as input and returns a new relation as output that contains a subset of the tuples of the input relation.

- That is, the output relation has the same number of columns as the input relation, but may have less rows.

To determine which tuples are in the output, the selection operation has a specified condition, called a ***predicate***, that tuples must satisfy to be in the output.

- The predicate is similar to a condition in an `if` statement.

# Selection Operation Formal Definition

The selection operation on relation *R* with predicate *F* is denoted by $\sigma_F(R)$.

$$\sigma_F(R)=\{t \mid t \in R \text{ and } F(t) \text{ is true}\}$$

where

- *R* is a relation, *t* is a tuple variable
- *F* is a formula (predicate) consisting of
  - ⇨operands that are constants or attributes
  - ⇨comparison operators: <, >, =, ≠, ≤, ≥
  - ⇨logical operators: AND, OR, NOT





# Selection Example

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

$\sigma_{title = 'EE'}(\text{Emp})$

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E6 | L. Chu | EE | 30000 |

$\sigma_{salary > 35000 \text{ OR } title = 'PR'}(\text{Emp})$

| eno | ename | title | salary |
|---|---|---|---|
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |





# Selection Question

***Question:*** Given this table and the query:

$\sigma_{salary > 50000 \text{ or } title='PR'}(\text{Emp})$

How many rows are returned?

**A)** 0
**B)** 1
**C)** 2
**D)** 3

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |





# Selection Question (2)

***Question:*** Given this table and the query:

$\sigma_{salary > 50000 \text{ or } title='PR'}(\text{Emp})$

How many columns are returned?

**A)** 0
**B)** 2
**C)** 3
**D)** 4

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |





# Selection Questions

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

Write the relational algebra expression that:
1) Returns all rows with an employee working on project P2.
2) Returns all rows with an employee who is working as a manager on a project.
3) Returns all rows with an employee working as a manager for more than 40 months.

Show the resulting relation for each case.





# Projection Operation

The ***projection operation*** is a unary operation that takes in a relation as input and returns a new relation as output that contains a subset of the attributes of the input relation and all non-duplicate tuples.

- The output relation has the same number of tuples as the input relation unless removing the attributes caused duplicates to be present.
- Question: When are we guaranteed to never have duplicates when performing a projection operation?

Besides the relation, the projection operation takes as input the names of the attributes that are to be in the output relation.

# Projection Operation Formal Definition

The projection operation on relation $R$ with output attributes $A_1,\ldots,A_m$ is denoted by $\Pi_{A_1,\ldots,A_m}(R)$.

$$\Pi_{A_1,\ldots,A_m}(R)=\{t[A_1,\ldots, A_m] \mid t \in R\}$$

where

- $R$ is a relation, $t$ is a tuple variable
- $\{A_1,\ldots,A_m\}$ is a subset of the attributes of $R$ over which the projection will be performed.
- Order of $A_1,\ldots, A_m$ is significant in the result.
- Cardinality of $\Pi_{A_1,\ldots,A_m}(R)$ is not necessarily the same as R because of duplicate removal.





# Projection Example

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

$\Pi_{eno,ename}$ (Emp)

| eno | ename |
|---|---|
| E1 | J. Doe |
| E2 | M. Smith |
| E3 | A. Lee |
| E4 | J. Miller |
| E5 | B. Casey |
| E6 | L. Chu |
| E7 | R. Davis |
| E8 | J. Jones |

$\Pi_{title}$ (Emp)

| title |
|---|
| EE |
| SA |
| ME |
| PR |





# Projection Question

***Question:*** Given this table and the query:

$\Pi_{title}$ (Emp)

How many rows are returned?

**A)** 0
**B)** 2
**C)** 4
**D)** 8

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |





# Projection Questions

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

Write the relational algebra expression that:
1) Returns only attributes *resp* and *dur*.
2) Returns only *eno*.
3) Returns only *pno*.

Show the resulting relation for each case.





# Union

***Union*** is a binary operation that takes two relations $R$ and $S$ as input and produces an output relation that includes all tuples that are either in $R$ or in $S$ or in both $R$ and $S$. Duplicate tuples are eliminated.

General form:

$$R \cup S = \{t \mid t \in R \text{ or } t \in S\}$$

where $R$, $S$ are relations, $t$ is a tuple variable.

$R$ and $S$ must be union-compatible. To be ***union-compatible*** means that the relations must have the same number of attributes with the same domains.

- Note that attribute names in both relations do not have to be the same. Result has attributes names of first relation.





# Union Example

Emp

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

WorksOn

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |

$\Pi_{eno}(\text{Emp}) \cup \Pi_{eno}(\text{WorksOn})$

| eno |
|---|
| E1 |
| E2 |
| E3 |
| E4 |
| E5 |
| E6 |
| E7 |
| E8 |

# Set Difference

***Set difference*** is a binary operation that takes two relations *R* and *S* as input and produces an output relation that contains all the tuples of *R* that are not in *S*.

General form:

$$R - S = \{t \mid t \in R \text{ and } t \notin S\}$$

where *R* and *S* are relations, *t* is a tuple variable.

Note that:

- R – S ≠ S – R
- R and S must be union compatible.





# Set Difference Example

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |

$\Pi_{eno}(\text{Emp}) - \Pi_{eno}(\text{WorksOn})$

| eno |
|---|
| E4 |
| E8 |

Question: What is the meaning of this query?

Question: What is $\Pi_{eno}(\text{WorksOn}) - \Pi_{eno}(\text{Emp})$?





# Intersection

***Intersection*** is a binary operation that takes two relations *R* and *S* as input and produces an output relation which contains all tuples that are in both *R* and *S*.

General form:

$$R \cap S = \{t \mid t \in R \text{ and } t \in S\}$$

where *R*, *S* are relations, *t* is a tuple variable.

- *R* and *S* must be union-compatible.

Note that $R \cap S = R - (R - S) = S - (S - R)$.





# Intersection Example

Emp

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

WorksOn

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |

$\Pi_{eno}(\text{Emp}) \cap \Pi_{eno}(\text{WorksOn})$

| eno |
|---|
| E1 |
| E2 |
| E3 |
| E5 |
| E6 |
| E7 |





# Set Operations Union-compatible Question

***Question:*** Two tables have the same number of fields in the same order with the same types, but the names of some fields are different. **True or false:** The two tables are union-compatible.

**A)** true

**B)** false





# Cartesian Product

The ***Cartesian product*** of two relations *R* (of degree $k_1$) and S (of degree $k_2$) is:

$$R \times S = \{t \mid t[A_1,\ldots,A_{k_1}] \in R \text{ and } t[A_{k_1+1},\ldots,A_{k_1+k_2}] \in S\}$$

The result of $R \times S$ is a relation of degree $(k_1 + k_2)$ and consists of all $(k_1 + k_2)$-tuples where each tuple is a concatenation of one tuple of *R* with one tuple of *S*.

The cardinality of R × S is $|R| * |S|$.

The Cartesian product is also known as ***cross product***.

# Cartesian Product Example

*Emp* Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |

*Proj* Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |

*Emp* × *Proj*

| eno | ename | title | salary | pno | pname | budget |
|---|---|---|---|---|---|---|
| E1 | J. Doe | EE | 30000 | P1 | Instruments | 150000 |
| E2 | M. Smith | SA | 50000 | P1 | Instruments | 150000 |
| E3 | A. Lee | ME | 40000 | P1 | Instruments | 150000 |
| E4 | J. Miller | PR | 20000 | P1 | Instruments | 150000 |
| E1 | J. Doe | EE | 30000 | P2 | DB Develop | 135000 |
| E2 | M. Smith | SA | 50000 | P2 | DB Develop | 135000 |
| E3 | A. Lee | ME | 40000 | P2 | DB Develop | 135000 |
| E4 | J. Miller | PR | 20000 | P2 | DB Develop | 135000 |
| E1 | J. Doe | EE | 30000 | P3 | CAD/CAM | 250000 |
| E2 | M. Smith | SA | 50000 | P3 | CAD/CAM | 250000 |
| E3 | A. Lee | ME | 40000 | P3 | CAD/CAM | 250000 |
| E4 | J. Miller | PR | 20000 | P3 | CAD/CAM | 250000 |





# Cartesian Product Question

***Question:*** *R* is a relation with 10 rows and 5 columns. *S* is a relation with 8 rows and 3 columns.

What is the degree and cardinality of the Cartesian product?

**A)** degree = 8, cardinality = 80

**B)** degree = 80, cardinality = 8

**C)** degree = 15, cardinality = 80

**D)** degree = 8, cardinality = 18





# θ-Join

Theta (θ) join is a derivative of the Cartesian product. Instead of taking all combinations of tuples from *R* and *S*, we only take a subset of those tuples that match a given condition *F*:

$$R \bowtie_F S = \{t \mid t[A_1,\ldots,A_{k_1}] \in R \text{ and } t[A_{k_1+1},\ldots,A_{k_1+k_2}] \in S \text{ and } F(t) \text{ is true}\}$$

where

- *R*, *S* are relations, *t* is a tuple variable
- *F*(*t*) is a formula defined as that of selection.

Note that $R \bowtie_F S = \sigma_F(R \times S)$.





# θ-Join Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn $\bowtie_{dur*10000 > budget}$ Proj

| eno | pno | resp | dur | P.pno | pname | budget |
|---|---|---|---|---|---|---|
| E2 | P1 | Analyst | 24 | P1 | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | P2 | DB Develop | 135000 |
| E3 | P4 | Engineer | 48 | P1 | Instruments | 150000 |
| E3 | P4 | Engineer | 48 | P2 | DB Develop | 135000 |
| E3 | P4 | Engineer | 48 | P3 | CAD/CAM | 250000 |
| E3 | P4 | Engineer | 48 | P4 | Maintenance | 310000 |
| E5 | P2 | Manager | 24 | P1 | Instruments | 150000 |
| E5 | P2 | Manager | 24 | P2 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | P1 | Instruments | 150000 |
| E6 | P4 | Manager | 48 | P2 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | P3 | CAD/CAM | 250000 |
| E6 | P4 | Manager | 48 | P4 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | P1 | Instruments | 150000 |
| E7 | P3 | Engineer | 36 | P2 | DB Develop | 135000 |
| E7 | P3 | Engineer | 36 | P3 | CAD/CAM | 250000 |
| E7 | P4 | Engineer | 23 | P1 | Instruments | 150000 |
| E7 | P4 | Engineer | 23 | P2 | DB Develop | 135000 |





# Types of Joins

The θ-Join is a general join in that it allows any expression in the condition *F*. However, there are more specialized joins that are frequently used.

A ***equijoin*** only contains the equality operator (=) in formula *F*.

- e.g. WorksOn $\bowtie_{WorksOn.pno = Proj.pno}$ Proj

A ***natural join*** over two relations *R* and *S* denoted by $R \bowtie S$ is the equijoin of *R* and *S* over a set of attributes common to both *R* and *S*.

- It removes the "extra copies" of the join attributes.
- The attributes must have the same name in both relations.





# Equijoin Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn $\bowtie_{WorksOn.pno = Proj.pno}$ Proj

| eno | pno | resp | dur | P.pno | pname | budget |
|---|---|---|---|---|---|---|
| E1 | P1 | Manager | 12 | P1 | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | P1 | Instruments | 150000 |
| E2 | P2 | Analyst | 6 | P2 | DB Develop | 135000 |
| E3 | P4 | Engineer | 48 | P4 | Maintenance | 310000 |
| E5 | P2 | Manager | 24 | P2 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | P4 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | P3 | CAD/CAM | 250000 |
| E7 | P4 | Engineer | 23 | P4 | Maintenance | 310000 |

What is the meaning of this join?

# Natural join Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn ⋈ Proj

| eno | pno | resp | dur | pname | budget |
|---|---|---|---|---|---|
| E1 | P1 | Manager | 12 | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | Instruments | 150000 |
| E2 | P2 | Analyst | 6 | DB Develop | 135000 |
| E3 | P4 | Engineer | 48 | Maintenance | 310000 |
| E5 | P2 | Manager | 24 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | CAD/CAM | 250000 |
| E7 | P4 | Engineer | 23 | Maintenance | 310000 |

Natural join is performed by comparing *pno* in both relations.





# Join Practice Questions

Emp Relation

| eno | ename | title | salary |
|---|---|---|---|
| E1 | J. Doe | EE | 30000 |
| E2 | M. Smith | SA | 50000 |
| E3 | A. Lee | ME | 40000 |
| E4 | J. Miller | PR | 20000 |
| E5 | B. Casey | SA | 50000 |
| E6 | L. Chu | EE | 30000 |
| E7 | R. Davis | ME | 40000 |
| E8 | J. Jones | SA | 50000 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P5 | Engineer | 23 |
| E8 | P3 | Manager | 40 |

Compute the following joins (counts only):

1) Emp $\bowtie_{\text{title='EE' and budget > 400000}}$ Proj
2) Emp ⋈ WorksOn
3) Emp ⋈ WorksOn ⋈ Proj
4) $\text{Proj}_1 \bowtie_{\text{Proj1.budget > Proj2.budget}} \text{Proj}_2$





# Outer Joins

Outer joins are used in cases where performing a join "loses" some tuples of the relations. These are called *dangling tuples*.

There are three types of outer joins:

- 1) ***Left outer join*** - $R ⟕ S$ - The output contains all tuples of *R* that match with tuples of *S*. If there is a tuple in *R* that matches with no tuple in *S*, the tuple is included in the final result and is padded with nulls for the attributes of *S*.
- 2) ***Right outer join*** - $R ⟖ S$ - The output contains all tuples of *S* that match with tuples of *R*. If there is a tuple in *S* that matches with no tuple in *R*, the tuple is included in the final result and is padded with nulls for the attributes of *R*.
- 3) ***Full outer join*** - $R ⟗ S$ - All tuples of *R* and *S* are included in the result whether or not they have a matching tuple in the other relation.





# Right Outer Join Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

WorksOn $⟖_{\text{WorksOn.pno = Proj.pno}}$ Proj

| eno | pno | resp | dur | P.pno | pname | budget |
|---|---|---|---|---|---|---|
| E1 | P1 | Manager | 12 | P1 | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | P1 | Instruments | 150000 |
| E2 | P2 | Analyst | 6 | P2 | DB Develop | 135000 |
| E3 | P4 | Engineer | 48 | P4 | Maintenance | 310000 |
| E5 | P2 | Manager | 24 | P2 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | P4 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | P3 | CAD/CAM | 250000 |
| E7 | P4 | Engineer | 23 | P4 | Maintenance | 310000 |
| null | null | null | null | P5 | CAD/CAM | 500000 |





# Outer Join Question

***Question:*** Given this table and the query:

WorksOn $⟗_{\text{WorksOn.pno = Proj.pno}}$ Proj

How many rows are returned?

**A)** 10
**B)** 9
**C)** 8
**D)** 7

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P4 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |





# Semi-Join and Anti-Join

A ***semi-join*** between tables returns rows from the first table where one or more matches are found in the second table.

- Semi-joins are used in `EXISTS` and `IN` constructs in SQL.

An ***anti-join*** between two tables returns rows from the first table where ***no*** matches are found in the second table.

- Anti-joins are used with `NOT EXISTS`, `NOT IN`, and `FOR ALL`.
- Anti-join is the complement of semi-join: $R ▷ S = R - R ◁ S$

# Semi-Join Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj $\ltimes_{Proj.pno = WorksOn.pno}$ WorksOn

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |





# Anti-Join Example

WorksOn Relation

| eno | pno | resp | dur |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P4 | Engineer | 48 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |
| E7 | P4 | Engineer | 23 |

Proj $\triangleright_{Proj.pno = WorksOn.pno}$ WorksOn

| pno | pname | budget |
|---|---|---|
| P5 | CAD/CAM | 500000 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | CAD/CAM | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |





# Aside: Division Operator

There is also a division operator used when you want to determine if *all* combinations of a relationship are present.

- E.g. Return the list of employees who work on *all* the projects that 'John Smith' works on.

The division operator is not a base operator and is not frequently used, so we will not spend any time on it.

- Note that $R \div S = \Pi_{R-S}(R) - \Pi_{R-S}((\Pi_{R-S}(R) \times S) - R)$.





# Combining Operations

Relational algebra operations can be combined in one expression by nesting them:

$$\Pi_{eno,pno,dur}(\sigma_{ename='J.\ Doe'}(Emp) \bowtie \sigma_{dur>16}(WorksOn))$$

- Return the eno, pno, and duration for employee 'J. Doe' when he has worked on a project for more than 16 months.

Operations also can be combined by using temporary relation variables to hold intermediate results.

- We will use the assignment operator $\leftarrow$ for indicating that the result of an operation is assigned to a temporary relation.

$$empdoe \leftarrow \sigma_{ename='J.\ Doe'}(Emp)$$
$$wodur \leftarrow \sigma_{dur>16}(WorksOn)$$
$$empwo \leftarrow empdoe \bowtie wodur$$
$$result \leftarrow \Pi_{eno,pno,dur}(empwo)$$





# Rename Operation

Renaming can be applied when assigning a result:

$$result(EmployeeNum, ProjectNum, Duration) \leftarrow \Pi_{eno,pno,dur}(empwo)$$

Or by using the rename operator $\rho$ (rho):

$$\rho_{result(EmployeeName, ProjectNum, Duration)}(empwo)$$





# Operator Precedence

Just like mathematical operators, the relational operators have precedence.

The precedence of operators from highest to lowest is:

- unary operators - $\sigma$, $\Pi$, $\rho$
- Cartesian product and joins - X, $\bowtie$
- intersection, division
- union and set difference

Parentheses can be used to changed the order of operations.

Note that there is no universal agreement on operator precedence, so we ***always*** use parentheses around the argument for both unary and binary operators.

## Complete Set of Relational Algebra Operators

It has been shown that the relational operators {σ, Π, ×, ∪, -} form a complete set of operators.

- That is, any of the other operators can be derived from a combination of these 5 basic operators.

Examples:

- Intersection - $R \cap S \equiv R \cup S - ((R - S) \cup (S - R))$
- We have also seen how a join is a combination of a Cartesian product followed by a selection.





## Relational Algebra Query Examples

Consider the database schema

Emp (eno, ename, title, salary)
Proj (pno, pname, budget)
WorksOn (eno, pno, resp, dur)

Queries:

- List the names of all employees.
  - ⇨ $\Pi_{ename}(Emp)$
- Find the names of projects with budgets over $100,000.
  - ⇨ $\Pi_{pname}(\sigma_{budget>100000}(Proj))$





## Practice Questions

Relational database schema:

branch (bname, address, city, assets)
customer (cname, street, city)
deposit (accnum, cname, bname, balance)
borrow (accnum, cname, bname, amount)

1) List the names of all branches of the bank.

2) List the names of all deposit customers together with their account numbers.

3) Find all cities where at least one customer lives.

4) Find all cities with at least one branch.

5) Find all cities with at least one branch or customer.

6) Find all cities that have a branch but no customers who live in that city.





## Practice Questions (2)

branch (bname, address, city, assets)
customer (cname, street, city)
deposit (accnum, cname, bname, balance)
borrow (accnum, cname, bname, amount)

1) Find the names of all branches with assets greater than $2,500,000.

2) List the name and cities of all customers who have an account with balance greater than $2,000.

3) List all the cities with at least one customer but without any bank branches.

4) Find the name of all the customers who live in a city with no bank branches.





## Practice Questions (3)

branch (bname, address, city, assets)
customer (cname, street, city)
deposit (accnum, cname, bname, balance)
borrow (accnum, cname, bname, amount)

1) Find all the cities that have both customers and bank branches.

2) List the customer name and loan and deposit amounts, who have a loan larger than a deposit account at the same branch.

3) Find the name and assets of all branches which have deposit customers living in Vancouver.

4) Find all the customers who have both a deposit account and a loan at the branch with name CalgaryCentral.

5) Your own?





## Other Relational Algebra Operators

There are other relational algebra operators that we will not discuss. Most notably, we often need ***aggregate operations*** that compute functions on the data.

For example, given the current operators, we cannot answer the query:

- What is the total amount of deposits at the Kelowna branch?

We will see how to answer these queries when we study SQL.

## Conclusion

The ***relational model*** represents data as relations which are sets of tuples. Each relational schema consists of a set of attribute names which represent a domain.

The relational model has several forms of ***constraints*** to guarantee data integrity including:

- domain, entity integrity and referential integrity constraints

***Keys*** are used to uniquely identify tuples in relations.

***Relational algebra*** is a set of operations for answering queries on data stored in the relational model.

- The 5 basic relational operators are: {$\sigma$, $\Pi$, $\times$, $\cup$, -}.
- By combining relational operators, queries can be answered over the base relations.





## Objectives

- Define: relation, attribute, tuple, domain, degree, cardinality, relational DB, intension, extension
- Define: relation schema, relational database schema, relation instance, null
- Perform Cartesian product given two sets.
- List the properties of relations.
- Define: superkey, key, candidate key, primary key, foreign key
- Define: integrity, constraints, domain constraint, entity integrity constraint, referential integrity constraint
- Given a relation be able to:
  - identify its cardinality, degree, domains, keys, and superkeys
  - determine if constraints are being violated





## Objectives (2)

- Define: relational algebra, query language
- Define and perform all relational algebra operators.
- List the operators which form the complete set of operators.
- Show how other operators can be derived from the complete set.

Given a relational schema and instance be able to translate English queries into relational algebra and show the resulting relation.

# COSC 304
# Introduction to Database Systems

## SQL DDL

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## SQL Overview

Structured Query Language or SQL is the standard query language for relational databases.

- It first became an official standard in 1986 as defined by the American National Standards Institute (ANSI).
- All major database vendors conform to the SQL standard with minor variations in syntax (different *dialects*).
- SQL consists of both a Data Definition Language (DDL) and a Data Manipulation Language (DML).

SQL is a ***declarative language*** (non-procedural). A SQL query specifies *what* to retrieve but not *how* to retrieve it.

- Basic SQL is not a complete programming language as it does not have control or iteration commands.
  - ⇨Procedural extensions: PL/SQL (Oracle), T-SQL (SQL Server)





## SQL History

- 1970 - Codd invents relational model and relational algebra
- 1974 - D. Chamberlin (also at IBM) defined Structured English Query Language (SEQUEL)
- 1976 - SEQUEL/2 defined and renamed SQL for legal reasons.
  - ⇨Origin of pronunciation 'See-Quel' but official pronunciation is 'S-Q-L'.
- Late 1970s - System R, Oracle, INGRES implement variations of SQL-like query languages.
- 1982 - standardization effort on SQL begins
- 1986 - became ANSI official standard
- 1987 - became ISO standard
- 1992 - SQL2 (SQL92) revision
- 1999 - SQL3 (supports recursion, object-relational)
- Updates: SQL:2003, SQL:2006, SQL:2008, SQL:2011,SQL:2016





## SQL Basic Rules

Some basic rules for SQL statements:

- 1) There is a set of *reserved words* that cannot be used as names for database objects. (e.g. `SELECT`, `FROM`, `WHERE`)
- 2) SQL is *case-insensitive*.
  - ⇨Only exception is string constants. 'FRED' not the same as 'fred'.
- 3) SQL is *free-format* and white-space is ignored.
- 4) The semi-colon is often used as a statement terminator, although that is not always required.
- 5) Date and time constants have defined format:
  - ⇨Dates: 'YYYY-MM-DD' e.g. '1975-05-17'
  - ⇨Times: 'hh:mm:ss[.f] ' e.g. '15:00:00'
  - ⇨Timestamp: 'YYYY-MM-DD hh:mm:ss[.f] ' e.g. '1975-05-17 15:00:00'
- 6) Two single quotes '' are used to represent a single quote character in a character constant. e.g. 'Master''s'.





## SQL DDL Overview

SQL contains a data definition language (DDL) that allows users to:

- add, modify, and drop tables
- create views
- define and enforce integrity constraints
- enforce security restrictions





## SQL Identifiers

***Identifiers*** are used to identify objects in the database such as tables, views, and columns.

- The identifier is the name of the database object.

An SQL identifier (name) must follow these rules:

- only contain upper or lower case characters, digits, and underscore ("_") character
- be no longer than 128 characters
  - ⇨DB vendors may impose stricter limits than this.
- must start with a letter (or underscore)
- cannot contain spaces
- Note: Quoted or ***delimited identifiers*** enclosed in double quotes allow support for spaces and other characters. E.g. `"select"`

## Database Identifier Question

***Question:*** Select ***one*** valid identifier.

**A)** 23test

**B)** 'fred'

**C)** test_!

**D)** field_

**E)** from





## Delimited Database Identifiers

***Question:*** **True or False:** `"from"` can be used as a valid identifier according to the SQL standard.

**A)** True

**B)** False





## SQL Data Types

In the relational model, each attribute has an associated *domain* of values.

In SQL, each column (attribute) has a ***data type*** that limits the values that it may store. The standard SQL data types are similar to their programming language equivalents.

The database will perform (implicit) data type conversion when necessary.

Explicit data type conversion using functions such as `CAST` and `CONVERT`.





## SQL Data Types (2)

| Data Type | Description |
|---|---|
| BOOLEAN | TRUE or FALSE |
| CHAR | Fixed length string (padded with blanks) e.g. CHAR(10) |
| VARCHAR | Variable length string e.g. VARCHAR(50) |
| BIT | Bit string e.g. BIT(4) can store '0101' |
| NUMERIC or DECIMAL | Exact numeric data type e.g. NUMERIC(7,2) has a precision (max. digits) of 7 and scale of 2 (# of decimals) e.g. 12345.67 |
| INTEGER | Integer data only |
| SMALLINT | Smaller space than INTEGER |
| FLOAT or REAL | Approximate numeric data types. |
| DOUBLE PRECISION | Precision dependent on implementation. |
| DATE | Stores YEAR, MONTH, DAY |
| TIME | Stores HOUR, MINUTE, SECOND |
| TIMESTAMP | Stores date and time data. |
| INTERVAL | Time interval. |
| CHARACTER LARGE OBJECT | Stores a character array (e.g. for a document) |
| BINARY LARGE OBJECT | Stores a binary array (e.g. for a picture, movie) |





## SQL User Defined Data Types

The **`CREATE DOMAIN`** command allows you to define your own types that are subsets of built-in types:

```
CREATE DOMAIN domainName AS dataType
   [DEFAULT defaultValue]
   [CHECK (condition)]
```

Example: Create user-defined domain for `Emp.title`:

```
CREATE DOMAIN titleType AS CHAR(2)
   DEFAULT 'EE'
   CHECK (VALUE IN (NULL,'EE','SA','PR','ME'));
```





## SQL User Defined Data Types (2)

The **`CHECK`** clause can use a nested select statement to retrieve values from the database:

```
CREATE DOMAIN mgrType AS CHAR(5)
   DEFAULT NULL
   CHECK (VALUE IN (SELECT eno FROM emp
                    WHERE title = 'ME' OR title = 'SA'));
```

Domains can be removed from the system using **`DROP`**:

```
DROP DOMAIN domainName [RESTRICT | CASCADE]
```

- `RESTRICT` - if domain is currently used, drop fails.
- `CASCADE` - if domain is current used, domain dropped and fields using domain defaulted to base type.

◆Example:

```
DROP DOMAIN mgrType;
```

## SQL CREATE TABLE

The **CREATE TABLE** command is used to create a table in the database. A table consists of a table name, a set of fields with their names and data types, and specified constraints.

The general form is:

```
CREATE TABLE tableName (
   attr1Name attr1Type  [attr1_constraints],
   attr2Name attr2Type  [attr2_constraints],
   ...
   attrMName attrMType  [attrM_constraints],
   [primary and foreign key constraints]
);
```





## SQL CREATE TABLE Example

The CREATE TABLE command for the Emp relation:

```
CREATE TABLE Emp (
   eno      CHAR(5),
   ename    VARCHAR(30) NOT NULL,
   bdate    DATE,
   title    CHAR(2),
   salary   DECIMAL(9,2),
   supereno CHAR(5),
   dno      CHAR(5),
   PRIMARY KEY (eno),
   FOREIGN KEY (dno) REFERENCES Dept(dno)
         ON DELETE SET NULL ON UPDATE CASCADE
);
```





## SQL Constraints

Constraints are specified in CREATE and ALTER TABLE statements.

Types of constraints:

- 1) **Required data** - To specify that a column must always have a data value (cannot be NULL) specify NOT NULL after the column definition.
  - ⇨e.g. `eno CHAR(5) NOT NULL`
- 2) **Domain constraints** - Used to verify that the value of a column is in a given domain using CHECK.
  - ⇨e.g. `title CHAR(2) CHECK (title IN (NULL,'EE','SA','PR','ME'));`
  - ⇨Forces the title to be either NULL or one of 4 defined values.
  - ⇨Can also be performed using user-defined types (domains).





## SQL Constraints (2)

- 3) **Tuple constraints** - CHECK can also be used on an entire tuple instead of a single attribute:

```
CREATE TABLE student(
   num      CHAR(10)    NOT NULL,
   honors   CHAR(1),
   gpa      DECIMAL(3,2),
   CHECK ( (honors = 'Y' AND gpa > 3.50)
            OR honors = 'N')
);
```

- Note that CHECK clause can contain subqueries, but the CHECK is only performed when the relation itself is modified.
  - ⇨Does not perform check when relation involved in subquery is modified.





## SQL Constraints - Entity Integrity

**Entity Integrity constraint** - The primary key of a table must contain a unique, non-null value for each row. The primary key is specified using the PRIMARY KEY clause.

- e.g. `PRIMARY KEY (eno)` (for Emp relation)
- e.g. `PRIMARY KEY (eno,pno)` (for WorksOn relation)
- It is also possible to use PRIMARY KEY right after defining the attribute in the CREATE TABLE statement.

There can only be one primary key per relation, other candidate keys can be specified using UNIQUE:

- e.g. `UNIQUE (ename)`





## SQL Constraints - Referential Integrity

**Referential integrity constraint** - Defines a foreign key that references the primary key of another table.

- If a foreign key contains a value that is not NULL, that value must be present in some tuple in the relation containing the referenced primary key.

Example: WorksOn contains two foreign keys:

- `WorksOn.eno` references `Emp.eno`
- `WorksOn.pno` references `Proj.pno`

Specify foreign keys using FOREIGN KEY syntax:

```
FOREIGN KEY (eno) REFERENCES Emp(eno)
```

## SQL Referential Integrity Example

The CREATE TABLE command for the WorksOn relation:

```
CREATE TABLE WorksOn (
   eno   CHAR(5),
   pno   CHAR(5),
   resp  VARCHAR(20),
   hours SMALLINT,
   PRIMARY KEY (eno,pno),
   FOREIGN KEY (eno) REFERENCES Emp(eno),
   FOREIGN KEY (pno) REFERENCES Proj(pno)
);
```





## SQL Referential Integrity and Updates

When you try to INSERT or UPDATE a **row in a relation containing a foreign key** (e.g. WorksOn) that operation is rejected if it violates referential integrity.

When you UPDATE or DELETE a **row in the primary key relation** (e.g. Emp or Proj), you have the option on what happens to the values in the foreign key relation (WorksOn):

- 1) CASCADE - Delete (update) values in foreign key relation when primary key relation has rows deleted (updated).
- 2) SET NULL - Set foreign key fields to NULL when corresponding primary key relation row is deleted.
- 3) SET DEFAULT - Set foreign key values to their default value (if defined).
- 4) NO ACTION - Reject the request on the parent table.





## SQL Referential Integrity Example (2)

```
CREATE TABLE WorksOn (
   eno   CHAR(5),
   pno   CHAR(5),
   resp  VARCHAR(20),
   hours SMALLINT,
   PRIMARY KEY (eno,pno),
   FOREIGN KEY (eno) REFERENCES Emp(eno)
                             ON DELETE NO ACTION
                             ON UPDATE CASCADE,
   FOREIGN KEY (pno) REFERENCES Proj(pno)
                             ON DELETE NO ACTION
                             ON UPDATE CASCADE
);
```





## Enforcing Referential Integrity Question

***Question:*** Select ***one*** true statement.

**A)** SET NULL can be used for the WorksOn.eno foreign key.

**B)** ON UPDATE CASCADE will modify all rows in the primary key table when a value is modified in the foreign key table.

**C)** SET DEFAULT cannot be used for the WorksOn.eno foreign key.

**D)** If a primary key row is deleted and it is referenced by a foreign key row, NO ACTION will generate an error to the user.





## SQL *CREATE TABLE* Full Syntax

Full syntax of CREATE TABLE statement:

```
CREATE TABLE tableName (
  { attrName attrType  [NOT NULL] [UNIQUE] [PRIMARY KEY]
     [DEFAULT value] [CHECK (condition)] }
  [PRIMARY KEY (colList)]
  {[FOREIGN KEY (colList) REFERENCES tbl [(colList)],
     [ON UPDATE action]
     [ON DELETE action] }
  {[CHECK (condition)] }
);
```

used when matching to attributes that are not the primary key





## Creating the Example Database

```
CREATE DOMAIN T_eno AS CHAR(5);
CREATE DOMAIN T_pno AS CHAR(5);
CREATE DOMAIN T_dno AS CHAR(5);

CREATE TABLE Emp(
   eno      T_eno,
   ename    VARCHAR(30) NOT NULL,
   bdate    DATE,
   title    CHAR(2),
   salary   DECIMAL(9,2),
   supereno T_eno,
   dno      T_dno
   PRIMARY KEY (eno),
   FOREIGN KEY (dno) REFERENCES Dept(dno)
         ON DELETE SET NULL ON UPDATE CASCADE
);
```

## Creating the Example Database (2)

```
CREATE TABLE WorksOn (
   eno   T_eno,
   pno   T_pno,
   resp  VARCHAR(20),
   hours SMALLINT,
   PRIMARY KEY (eno,pno),
   FOREIGN KEY (eno) REFERENCES Emp(eno)
               ON DELETE NO ACTION ON UPDATE CASCADE,
   FOREIGN KEY (pno) REFERENCES Proj(pno)
               ON DELETE NO ACTION ON UPDATE CASCADE
);
```

Question:
Write `CREATE TABLE` statements to build the `Proj` and `Dept` relations:

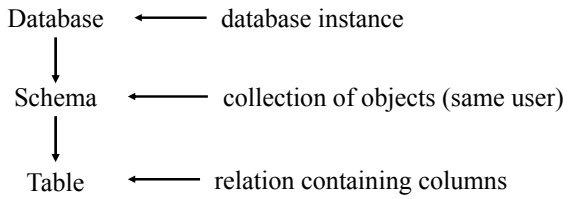
- `Dept(dno, dname, mgreno)`
- `Proj(pno, pname, budget, dno)`





## Defining a Database

There is typically a hierarchy of database objects that you can create, alter, and destroy.




SQL does not standardize how to create a database. A database often contains one or more *catalogs*, each of which contains a set of *schemas*.

⇨To make things more complicated, many DBMSs do not implement everything and rename things. e.g. A database *IS* a schema for MySQL (there is no `CREATE SCHEMA` command).





## Creating Schemas

A ***schema*** is a collection of database objects (tables, views, domains, etc.) usually associated with a single user.

Creating a schema: (User `Joe` creates the schema)

```
CREATE SCHEMA employeeSchema AUTHORIZATION Joe;
```

Dropping a schema:

```
DROP SCHEMA employeeSchema;
```





## ALTER TABLE

The **`ALTER TABLE`** command can be used to change an existing table. This is useful when the table already contains data and you want to add or remove a column or constraint.

⇨DB vendors may support only parts of `ALTER TABLE` or may allow additional changes including changing the data type of a column.

General form:

```
ALTER TABLE tableName
   [ADD [COLUMN] colName dataType [NOT NULL] [UNIQUE]
      [DEFAULT value] [CHECK (condition)] ]
   [DROP [COLUMN] colName [RESTRICT | CASCADE]
   [ADD [CONSTRAINT [constraintName]] constraintDef]
   [DROP CONSTRAINT constraintName [RESTRICT | CASCADE]]
   [ALTER [COLUMN] SET DEFAULT defValue]
   [ALTER [COLUMN] DROP DEFAULT]
```





## ALTER TABLE Examples

Add column `location` to `Dept` relation:

```
ALTER TABLE dept
  ADD location VARCHAR(50);
```

Add field `SSN` to `Emp` relation:

```
ALTER TABLE Emp
  ADD SSN CHAR(10);
```

Indicate that `SSN` is `UNIQUE` in `Emp`:

```
ALTER TABLE Emp
  ADD CONSTRAINT ssnConst UNIQUE(SSN);
```





## DROP TABLE

The command **`DROP TABLE`** is used to delete the table definition and all data from the database:

```
DROP TABLE tableName [RESTRICT | CASCADE];
```

Example:

```
DROP TABLE Emp;
```

Question: What would be the effect of the command:

```
DROP TABLE Emp CASCADE;
```

## Indexes

Indexes are used to speed up access to the rows of the tables based on the values of certain attributes.

- ⇨An index will often significantly improve the performance of a query, however they represent an overhead as they must be updated every time the table is updated.

The general syntax for creating and dropping indexes is:

```
CREATE [UNIQUE] INDEX indexName
   ON  tableName (colName [ASC|DESC] [,...])

DROP INDEX indexName;
```

- ◆UNIQUE means that each value in the index is unique.
- ◆ASC/DESC specifies the sorted order of index.





## Indexes Example

Creating an index on eno and pno in WorksOn is useful as it will speed up joins with the Emp and Proj tables respectively.

- ⇨Index is not UNIQUE as eno (pno) can occur many times in WorksOn.

```
CREATE INDEX idxEno ON WorksOn (eno);
CREATE INDEX idxPno ON WorksOn (pno);
```

Most DBMSs will put an index on the primary key, but if they did not, this is what it would like for WorksOn:

```
CREATE UNIQUE INDEX idxPK ON WorksOn (eno,pno);
```





## Database Updates

Database updates such as inserting rows, deleting rows, and updating rows are performed using their own statements.

Insert is performed using the INSERT command:

```
INSERT INTO tableName [(column list)]
VALUES (data value list)
```

Examples:

```
INSERT INTO emp VALUES ('E9','S. Smith',DATE '1975-03-05',
                        'SA',60000,'E8','D1');

INSERT INTO proj (pno, pname) VALUES ('P6','Programming');
```

Note: If column list is omitted, values must be specified in order they were created in the table. If any columns are omitted from the list, they are set to NULL.





## INSERT Multiple Rows

INSERT statement extended by many databases to take multiple rows:

```
INSERT INTO tableName [(column list)]
VALUES (data value list) [, (values) ]+
```

Example:

```
INSERT INTO Emp (eno, ename) VALUES
   ('E10', 'Fred'), ('E11', 'Jane'), ('E12', 'Joe')
```





## INSERT rows from SELECT

Insert multiple rows that are the result of a SELECT statement:

```
INSERT INTO tableName [(column list)]
  SELECT ...
```

Example: Add rows to a temporary table that contains only employees with title ='EE'.

```
INSERT INTO tmpTable
   SELECT eno, ename
   FROM emp
   WHERE title = 'EE'
```





## UPDATE Statement

Updating existing rows is performed using UPDATE statement:

```
UPDATE tableName
SET col1 = val1 [,col2=val2...]
[WHERE condition]
```

Examples:

- ◆1) Increase all employee salaries by 10%.

```
UPDATE emp SET salary = salary*1.10;
```

- ◆2) Increase salaries of employees in department 'D1' by 8%.

```
UPDATE emp SET salary = salary*1.08
WHERE dno = 'D1';
```

## *DELETE* Statement

Rows are deleted using the DELETE statement:

```
DELETE FROM tableName
[WHERE condition]
```

Examples:

- 1) Fire everyone in the company.

```
DELETE FROM workson;
DELETE FROM emp;
```

- 2) Fire everyone making over $35,000.

```
DELETE FROM emp
WHERE salary > 35000;
```





## Practice Questions

Relational database schema:

```
emp (eno, ename, bdate, title, salary,
supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)
```

1) Insert a department with number 'D5', name 'Useless', and no manager.
2) Insert a workson record with eno='E1' and pno='P3'.
3) Delete all records from emp.
4) Delete only the records in workson with more than 20 hours.
5) Update all employees to give them a 20% pay cut.
6) Update the projects for dno='D3' to increase their budget by 10%.





## Conclusion

SQL contains a data definition language that allows you to CREATE, ALTER, and DROP database objects such as tables, triggers, indexes, schemas, and views.

Constraints are used to preserve the integrity of the database:

- CHECK can be used to validate attribute values.
- **Entity Integrity constraint** - The primary key of a table must contain a unique, non-null value for each row.
- **Referential integrity constraint** - Defines a foreign key that references a unique key of another table.

INSERT, DELETE, and UPDATE commands modify the data stored within the database.





## Objectives

General:

- Recognize valid and invalid identifiers

Constraints:

- Define own domains using CREATE DOMAIN
- List 5 types of constraints and how to enforce them
  - ⇨required (not null) data, domain constraints, tuple constraints
  - ⇨entity integrity, referential integrity

Creating database objects:

- Describe hierarchy of database objects: database, schema, table
- Write CREATE TABLE statement given high-level description.
- List what ALTER TABLE can and cannot do.
- Be able to write INSERT, DELETE, and UPDATE commands.
- Create an index on fields of a table.

# COSC 304
# Introduction to Database Systems

# SQL

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## SQL Queries

Querying with SQL is performed using a `SELECT` statement. The general form of the statement is:

**SELECT** $A_1, A_2, \ldots, A_n$ ⟵ **attributes in result**
**FROM** $R_1, R_2, \ldots, R_m$ ⟵ **tables in query**
**WHERE** (*condition*)

Notes:

- 1) The "*" is used to select all attributes.
- 2) Combines the relational algebra operators of selection, projection, and join into a single statement.
- 3) Comparison operators: =, !=, >, <, >=, <=.





## SQL and Relational Algebra

The `SELECT` statement can be mapped directly to relational algebra.

**SELECT** $A_1, A_2, \ldots, A_n$
**FROM** $R_1, R_2, \ldots, R_m$
**WHERE** $P$

is equivalent to:

$$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$$





## Example Relations

Relations:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

Foreign keys:

- emp: emp.supereno to emp.eno, emp.dno to dept.dno
- proj: proj.dno to dept.dno
- dept: dept.mgreno to emp.eno
- workson: workson.eno to emp.eno, workson.pno to proj.pno





## Example Relation Instances

emp

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 1975-01-05 | EE | 30000 | E2 | null |
| E2 | M. Smith | 1966-06-04 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 1966-07-05 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 1950-09-01 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 1971-12-25 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 1965-11-30 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 1977-09-08 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 1972-10-11 | SA | 50000 | null | D1 |

workson

| eno | pno | resp | hours |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |

proj

| pno | pname | budget | dno |
|---|---|---|---|
| P1 | Instruments | 150000 | D1 |
| P2 | DB Develop | 135000 | D2 |
| P3 | Budget | 250000 | D3 |
| P4 | Maintenance | 310000 | D2 |
| P5 | CAD/CAM | 500000 | D2 |

dept

| dno | dname | mgreno |
|---|---|---|
| D1 | Management | E8 |
| D2 | Consulting | E7 |
| D3 | Accounting | E5 |
| D4 | Development | null |





## One Relation Query Example

Return the employee name and salary of all employees whose title is 'EE':

```
SELECT ename, salary
FROM   emp
WHERE  title = 'EE'
```

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

→

Result

| ename | salary |
|---|---|
| J. Doe | 30000 |
| L. Chu | 30000 |

Algorithm: Scan each tuple in table and check if matches condition in `WHERE` clause.

# One Relation Query Examples

Return the birth date and salary of employee 'J. Doe':

```
SELECT bdate, salary
FROM   emp
WHERE  ename = 'J. Doe'
```

Return all information on all employees:

```
SELECT *          ⟵ * returns all attributes
FROM   emp
```

Return the employee number, project number, and number of hours worked where the hours worked is > 50:

```
SELECT eno, pno, hours
FROM   workson
WHERE  hours > 50
```





# Duplicates in SQL

One major difference between SQL and relational algebra is that relations in SQL are ***bags*** instead of sets.

- It is possible to have two or more identical rows in a relation.

Consider the query: Return all titles of employees.

```
SELECT title
FROM   emp
```

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

Result

| title |
|---|
| EE |
| SA |
| ME |
| PR |
| SA |
| EE |
| ME |
| SA |





# Duplicates in SQL - *DISTINCT* clause

To remove duplicates, use the **DISTINCT** clause in the SQL statement:

```
SELECT DISTINCT title
FROM   emp
```

Result

| title |
|---|
| EE |
| SA |
| ME |
| PR |





# SQL Practice Questions Single Table

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) Return the project names that have a budget > 250000.

2) Return the employee numbers who make less than $30000.

3) Return the list of `workson` responsibilities (`resp`) with no duplicates.

4) Return the employee (names) born after July 1, 1970 that have a salary > 35000 and have a title of 'SA' or 'PR'.

- Write the equivalent relational algebra expression.





# Join Query Example

Multiple tables can be queried in a single SQL statement by listing them in the `FROM` clause.

- Note that if you do not specify any join condition to relate them in the `WHERE` clause, you get a *cross product* of the tables.

Example: Return the employees who are assigned to the 'Management' department.

```
SELECT ename
FROM   emp, dept
WHERE  dname = 'Management'
        and emp.dno = dept.dno
```

Result

| ename |
|---|
| R. Davis |
| J. Jones |





# Join Query Examples

Return the department names and the projects in each department:

```
SELECT dname, pname
FROM   dept, proj
WHERE  dept.dno = proj.dno
```

Return the employees and the names of their department:

```
SELECT ename, dname
FROM   emp, dept
WHERE  emp.dno=dept.dno
```

Return all projects who have an employee working on them whose title is 'EE':

```
SELECT pname
FROM   emp, proj, workson
WHERE  emp.title = 'EE' and workson.eno=emp.eno
        and workson.pno = proj.pno
```

## SQL Query Question

***Question:*** What query would return the name and salary of employees working on project `'P3'`:

**A)**
```
SELECT ename, salary
FROM emp, workson
WHERE emp.eno = workson.eno and pno = 'P3'
```

**B)**
```
SELECT ename, salary
FROM emp, workson, proj
WHERE emp.eno = workson.eno and pno = "P3"
```





## SQL Practice Questions Joins

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) For each employee, return their name and their department name.

2) Return the list of project names for the department with name `'Consulting'`.

3) Return `workson` records (eno, pno, resp, hours) where project budget is > $50000 and hours worked is < 20.

4) Return a list of all department names, the names of the projects of that department, and the name of the manager of each department.





## Calculated Fields

Mathematical expressions are allowed in the `SELECT` clause to perform simple calculations.

- When an expression is used to define an attribute, the DBMS gives the attribute a unique name such as `col1`, `col2`, etc.

Example: Return how much employee 'A. Lee' will get paid for his work on each project.

```
SELECT ename, pname, salary/52/5/8*hours
FROM   emp, workson, proj
WHERE  emp.eno = workson.eno and ename='A. Lee'
       and proj.pno = workson.pno
```

Result

| ename | pname | col3 |
|---|---|---|
| A. Lee | Budget | 192.31 |
| A. Lee | Maintenance | 923.08 |





## Renaming and Aliasing

Often it is useful to be able to rename an attribute in the final result (especially when using calculated fields). Renaming is accomplished using the keyword **`AS`**:

```
SELECT ename, pname, salary/52/5/8*hours AS pay
FROM   emp, workson, proj
WHERE  emp.eno = workson.eno and ename='A. Lee'
       and proj.pno = workson.pno
```

Result

| ename | pname | pay |
|---|---|---|
| A. Lee | Budget | 192.31 |
| A. Lee | Maintenance | 923.08 |

Note: AS keyword is optional.





## Renaming and Aliasing (2)

Renaming is also used when two or more copies of the same table are in a query. Using ***aliases*** allows you to uniquely identify what table you are talking about.

Example: Return the employees and their managers where the managers make less than the employee.

```
SELECT E.ename, M.ename
FROM   emp as E, emp as M
WHERE  E.supereno = M.eno and E.salary > M.salary
```





## Advanced Conditions - *BETWEEN*

Sometimes the condition in the `WHERE` clause will request tuples where one attribute value must be in a *range* of values.

Example: Return the employees who make at least $20,000 and less than or equal to $45,000.

```
SELECT ename
FROM   emp
WHERE  salary >= 20000 and salary <= 45000
```

We can use the keyword **`BETWEEN`** instead:

```
SELECT ename
FROM   emp
WHERE  salary BETWEEN 20000 and 45000
```

## Advanced Conditions - LIKE

For string valued attributes, the **LIKE** operator is used to search for partial matches.

- Partial string matches are specified by using either "%" that replaces an arbitrary number of characters or underscore "_" that replaces a single character.

Example: Return all employee names that start with 'A'.

```
SELECT ename
FROM   emp
WHERE  ename LIKE 'A%'
```

Example: Return all employee names who have a first name that starts with 'J' and whose last name is 3 characters long.

```
SELECT ename
FROM   emp
WHERE  ename LIKE 'J. ___'
```





## Performance Concerns of LIKE

**Warning**: Do not use the LIKE operator if you do not have to.

It is often an inefficient operation as the DBMS may not be able to optimize lookup using LIKE as it can for equal (=) comparisons. The result is the DBMS often has to examine ALL TUPLES in the relation.

In almost all cases, adding indexes will ***not*** increase the performance of LIKE queries because the indexes cannot be used.

- Most indexes are implemented using B-trees that allow for fast equality searching and efficient range searches.





## Advanced Conditions - IN

To specify that an attribute value should be in a given set of values, the **IN** keyword is used.

- Example: Return all employees who are in any one of the departments {'D1', 'D2', 'D3'}.

```
SELECT ename
FROM   emp
WHERE  dno IN ('D1','D2','D3')
```

Note that this is equivalent to using OR:

```
SELECT ename
FROM   emp
WHERE  dno = 'D1' OR dno = 'D2' OR dno = 'D3'
```

However, we will see more practical uses of IN and NOT IN when we study nested subqueries.





## Advanced Conditions - NULL

Remember NULL is used to indicate that a given attribute does not have a value. To determine if an attribute is NULL, we use the clause **IS NULL**.

- Note that you cannot test NULL values using = and <>.

Example: Return all employees who are not in a department.

```
SELECT ename
FROM   emp
WHERE  dno IS NULL
```

Example: Return all departments that have a manager.

```
SELECT dname
FROM   dept
WHERE  mgreno IS NOT NULL
```





## Set Operations

The set operations of union, intersection, and difference are used to combine the results of two SQL queries.

- UNION , INTERSECT, EXCEPT
- Note: UNION ALL returns all rows

Example: Return the employees who are either directly supervised by 'R. Davis' or directly supervised by 'M. Smith'.

```
(SELECT E.ename
FROM   emp as E, emp as M
WHERE  E.supereno = M.eno and M.ename='R. Davis')

UNION

(SELECT E.ename
FROM   emp as E, emp as M
WHERE  E.supereno = M.eno and M.ename='M. Smith')
```





## SELECT INTO

The result of a select statement can be stored in a temporary table using the **INTO** keyword.

```
SELECT E.ename
INTO   davisMgr
FROM   emp as E, emp as M
WHERE  E.supereno = M.eno and M.ename='R. Davis'
```

This can be used for set operations instead of using parentheses:

```
SELECT E.ename
INTO   smithMgr
FROM   emp as E, emp as M
WHERE  E.supereno = M.eno and M.ename='M. Smith'
davisMgr UNION smithMgr;
```

## Ordering Result Data

The query result returned is not ordered on any attribute by default. We can order the data using the **ORDER BY** clause:

```
SELECT   ename, salary, bdate
FROM     emp
WHERE    salary > 30000
ORDER BY salary DESC, ename ASC
```

- 'ASC' sorts the data in ascending order, and 'DESC' sorts it in descending order. The default is 'ASC'.
- The order of sorted attributes is significant. The first attribute specified is sorted on first, then the second attribute is used to break any ties, etc.
- NULL is normally treated as less than all non-null values.





## LIMIT and OFFSET

If you only want the first *N* rows, use a **LIMIT** clause:

```
SELECT   ename, salary FROM emp ORDER BY salary DESC
LIMIT 5
```

To start from a row besides the first, use **OFFSET**:

```
SELECT   eno, salary FROM emp ORDER BY eno DESC
LIMIT 3  OFFSET 2
```

- LIMIT improves performance by reducing amount of data processed and sent by the database system.
- OFFSET 0 is first row, so OFFSET 2 would return the 3rd row.
- LIMIT/OFFSET syntax supported differently by systems.
  - ⇨MySQL, PostgreSQL – use LIMIT syntax
  - ⇨Oracle – uses ROWNUM field that can be filtered in WHERE
  - ⇨SQL Server – use SELECT TOP *N*





## SQL Querying with NULL and LIKE

***Question:*** What query would return the department names that do not have a manager or contain 'ent'.

**A)**
```
SELECT dname
FROM   dept
WHERE mgreno = NULL OR dname LIKE '_ent'
```

**B)**
```
SELECT dname
FROM   dept
WHERE mgreno IS NULL OR dname LIKE '%ent%'
```





## SQL Practice Questions Expressions, LIKE, IS NULL

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) Calculate the monthly salary for each employee.

2) List all employee names who do not have a supervisor.

3) List all employee names where the employee's name contains an 'S' and workson responsibility that ends in 'ER'.

4) Return the list of employees (names) who make less than their managers and how much less they make.

5) Return only the top 3 project budgets in descending order.





## SQL Practice Questions Set Operations, ORDER BY

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) Return the list of employees sorted by salary (desc) and then title (asc).

2) Return the employees (names) who either manage a department or manage another employee.

3) Return the employees (names) who manage an employee but do not manage a department.

4) Give a list of all employees who work on a project for the 'Management' department ordered by project number (asc).

5) **Challenge:** Return the projects (names) that have their department manager working on them.





## Aggregate Queries and Functions

Several queries cannot be answered using the simple form of the SELECT statement. These queries require a summary calculation to be performed. Examples:

- What is the maximum employee salary?
- What is the total number of hours worked on a project?
- How many employees are there in department 'D1'?

To answer these queries requires the use of aggregate functions. These functions operate on a single column of a table and return a single value.

## Aggregate Functions

The five basic aggregate functions are:

- COUNT - returns the # of values in a column
- SUM - returns the sum of the values in a column
- AVG - returns the average of the values in a column
- MIN - returns the smallest value in a column
- MAX - returns the largest value in a column

Notes:

- 1) COUNT, MAX, and MIN apply to all types of fields, whereas SUM and AVG apply to only numeric fields.
- 2) Except for COUNT(*) all functions ignore nulls. COUNT(*) returns the number of rows in the table.
- 3) Use DISTINCT to eliminate duplicates.





## Aggregate Function Example

Return the number of employees and their average salary.

```
SELECT COUNT(eno) AS numEmp, AVG(salary) AS avgSalary
FROM   emp
```

Result

| numEmp | avgSalary |
|---|---|
| 8 | 38750 |





## GROUP BY Clause

Aggregate functions are most useful when combined with the GROUP BY clause. The **GROUP BY** clause groups the tuples based on the values of the attributes specified.

When used in combination with aggregate functions, the result is a table where each tuple consists of unique values for the group by attributes and the result of the aggregate functions applied to the tuples of that group.





## GROUP BY Example

For each employee title, return the number of employees with that title, and the minimum, maximum, and average salary.

```
SELECT   title, COUNT(eno) AS numEmp,
         MIN(salary) as minSal,
         MAX(salary) as maxSal, AVG(salary) AS avgSal
FROM     emp
GROUP BY title
```

Result

| title | numEmp | minSal | maxSal | avgSal |
|---|---|---|---|---|
| EE | 2 | 30000 | 30000 | 30000 |
| SA | 3 | 50000 | 50000 | 50000 |
| ME | 2 | 40000 | 40000 | 40000 |
| PR | 1 | 20000 | 20000 | 20000 |





## GROUP BY Clause Rules

There are a few rules for using the GROUP BY clause:

- 1) A column name cannot appear in the SELECT part of the query unless it is part of an aggregate function or in the list of group by attributes.
  - ⇨ Note that the reverse is allowed: a column can be in the GROUP BY without being in the SELECT part.
- 2) Any WHERE conditions are applied before the GROUP BY and aggregate functions are calculated.





## HAVING Clause

The **HAVING** clause is applied ***AFTER*** the GROUP BY clause and aggregate functions are calculated.

It is used to filter out entire *groups* that do not match certain criteria.

The **HAVING** clause can contain any condition that references aggregate functions and the group by attributes themselves.

- However, any conditions on the GROUP BY attributes should be specified in the WHERE clause if possible due to performance reasons.

## *HAVING* Example

Return the title and number of employees of that title where the number of employees of the title is at least 2.

```
SELECT   title, COUNT(eno) AS numEmp
FROM     emp
GROUP BY title
HAVING   COUNT(eno) >= 2
```

Result

| title | numEmp |
|---|---|
| EE | 2 |
| SA | 3 |
| ME | 2 |





## *GROUP BY/HAVING* Example

For employees born after December 1, 1965, return the average salary by department where the average is > 40,000.

```
SELECT   dname, AVG(salary) AS avgSal
FROM     emp, dept
WHERE    emp.dno = dept.dno and
           emp.bdate > DATE '1965-12-01'
GROUP BY dname
HAVING   AVG(salary) > 40000
```

Step #1: Perform Join and Filter in `WHERE` clause

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E2 | M. Smith | 1966-06-04 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 1966-07-05 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E5 | B. Casey | 1971-12-25 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E7 | R. Davis | 1977-09-08 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 1972-10-11 | SA | 50000 | null | D1 | Management | E8 |





## *GROUP BY/HAVING* Example (2)

Step #2: `GROUP BY` on `dname`

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E2 | M. Smith | 1966-06-04 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E5 | B. Casey | 1971-12-25 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E3 | A. Lee | 1966-07-05 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 1977-09-08 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 1972-10-11 | SA | 50000 | null | D1 | Management | E8 |

Step #3: Calculate aggregate functions

| dname | avgSal |
|---|---|
| Accounting | 50000 |
| Consulting | 40000 |
| Management | 45000 |

Step #4: Filter groups using `HAVING` clause

| dname | avgSal |
|---|---|
| Accounting | 50000 |
| Management | 45000 |





## *GROUP BY* Examples

Return the average budget per project:

```
SELECT AVG(budget)
FROM   proj
```

Return the average # of hours worked on each project:

```
SELECT   pno, AVG(hours)
FROM     workson
GROUP BY pno
```

Return the departments that have projects with at least 2 'EE's working on them:

```
SELECT   DISTINCT proj.dno
FROM     proj, workson, emp
WHERE    emp.title = 'EE' and workson.eno=emp.eno
           and workson.pno = proj.pno
GROUP BY proj.dno, proj.pno
HAVING   COUNT(*) >=2
```





## *GROUP BY/HAVING* Multi-Attribute Example

Return the employee number, department number and hours the employee worked per department where the hours is >= 10.

```
SELECT   W.eno, D.dno, SUM(hours)
FROM     workson AS W, dept AS D, proj AS P
WHERE    W.pno = P.pno and P.dno = D.dno
GROUP BY W.eno, D.dno
HAVING   SUM(hours) >= 10
```

Result:

| eno | dno | SUM(hours) |
|---|---|---|
| E1 | D1 | 12 |
| E2 | D1 | 24 |
| E3 | D2 | 48 |
| E3 | D3 | 10 |
| E4 | D2 | 18 |
| E5 | D2 | 24 |
| E6 | D2 | 48 |
| E7 | D3 | 36 |

Question:
1) How would you only return records for departments D2 and D3?





## SQL Querying with *GROUP BY*

***Question:*** Of the following queries, select one which is invalid.

**A)**
```
SELECT dname
FROM   dept
GROUP BY dno
```

**B)**
```
SELECT COUNT(*)
FROM   dept
```

**C)**
```
SELECT dno, COUNT(*)
FROM   dept
```

**D)**
```
SELECT dno, COUNT(*)
FROM   dept  WHERE mgreno > 'A'
GROUP BY dno, dname
```

## `GROUP BY` Practice Questions

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) Return the highest salary of any employee.

2) Return the smallest project budget.

3) Return the department number and average budget for its projects.

4) For each project, return its name and the total number of hours employees have worked on it.

5) For each employee, return the total number of hours they have worked. Only show employees with more than 30 hours.





## Subqueries

SQL allows a single query to have multiple subqueries nested inside of it. This allows for more complex queries to be written.

When queries are nested, the outer statement determines the contents of the final result, while the inner `SELECT` statements are used by the outer statement (often to lookup values for `WHERE` clauses).

A subquery can be in the `SELECT`, `FROM`, `WHERE` or `HAVING` clause.

```
SELECT   ename, salary, bdate
FROM     emp
WHERE    salary > (SELECT AVG(salary) FROM emp)
```





## Types of Subqueries

There are three types of subqueries:

- 1) ***scalar subqueries*** - return a single value. Often value is then used in a comparison.
  - ⇨ If query is written so that it expects a subquery to return a single value, and if it returns multiple values or no values, a run-time error occurs.
- 2) ***row subquery*** - returns a single row which may have multiple columns.
- 3) ***table subquery*** - returns one or more columns and multiple rows.





## Scalar Subquery Examples

Return the employees that are in the 'Accounting' department:

```
SELECT ename
FROM   emp
WHERE  dno = (SELECT dno FROM dept
              WHERE dname = 'Accounting')
```

Return all employees who work more hours than average on a single project:

```
SELECT ename
FROM   emp, workson
WHERE  workson.eno = emp.eno AND
       workson.hours > (SELECT AVG(hours) FROM workson)
```





## Table Subqueries

A table subquery returns a relation. There are several operators that can be used:

- `EXISTS` *R* - true if *R* is not empty
- *s* `IN` *R* - true if *s* is equal to one of the values of *R*
- *s* > `ALL` *R* - true if *s* is greater than **every** value in *R*
- *s* > `ANY` *R* - true if *s* is greater than **any** value in *R*

Notes:

- 1) Any of the comparison operators (<, <=, =, etc.) can be used.
- 2) The keyword `NOT` can proceed any of the operators.
  - ⇨ Example: *s* `NOT IN` *R*





## Table Subquery Examples

Return all departments who have a project with a budget greater than $300,000:

```
SELECT dname FROM dept WHERE dno IN
(SELECT dno FROM proj WHERE budget > 300000)
```

Return all projects that 'J. Doe' works on:

```
SELECT pname FROM proj WHERE pno IN
      (SELECT pno FROM workson WHERE eno =
         (SELECT eno FROM emp WHERE ename = 'J. Doe'))
```

## *EXISTS* Example

The EXISTS function is used to check whether the result of a nested query is empty or not.

- EXISTS returns true if the nested query has 1 or more tuples.

Example: Return all employees who have the same name as someone else in the company.

```
SELECT ename
FROM   emp as E
WHERE  EXISTS (SELECT * FROM emp as E2
               WHERE E.ename = E2.ename AND
                     E.eno <> E2.eno)
```





## *ANY* and *ALL* Example

ANY means that any value returned by the subquery can satisfy the condition.

ALL means that all values returned by the subquery must satisfy the condition.

Example: Return the employees who make more than all the employees with title 'ME' make.

```
SELECT ename
FROM   emp as E
WHERE  salary > ALL (SELECT salary FROM emp
                     WHERE title = 'ME')
```





## Subquery Syntax Rules

1) The ORDER BY clause may not be used in a subquery.

2) The number of attributes in the SELECT clause in the subquery must match the number of attributes compared to with the comparison operator.

3) Column names in a subquery refer to the table name in the FROM clause of the subquery by default. You must use aliasing if you want to access a table that is present in both the inner and outer queries.





## Correlated Subqueries

Most queries involving subqueries can be rewritten so that a subquery is not needed.

- This is normally beneficial because query optimizers may not do a good job at optimizing queries containing subqueries.

A nested query is ***correlated*** with the outside query if it must be re-computed for every tuple produced by the outside query. Otherwise, it is ***uncorrelated***, and the nested query can be converted to a non-nested query using joins.

A nested query is correlated with the outer query if it contains a reference to an attribute in the outer query.





## Correlated Subquery Example

Return all employees who have the same name as another employee:

```
SELECT ename
FROM   emp as E
WHERE  EXISTS (SELECT eno FROM emp as E2
               WHERE E.ename = E2.ename AND
                     E.eno <> E2.eno)
```

A more efficient solution with joins:

```
SELECT E.ename
FROM   emp as E, emp as E2
WHERE  E.ename = E2.ename AND E.eno <> E2.eno
```





## Explicit Join Syntax

You can cpecify a join condition directly in the FROM clause instead of the WHERE.

Example #1: Return the employees who are assigned to the 'Management' department:

```
SELECT ename
FROM   emp JOIN dept ON emp.dno = dept.dno
WHERE  dname = 'Management'
```

Example #2: Return all projects who have an employee working on them whose title is 'EE':

```
SELECT pname
FROM   emp E JOIN workson W ON E.eno = W.eno
          JOIN proj AS P ON W.pno = P.pno
WHERE  E.title = 'EE'
```

## Outer Joins

Using joined tables in the FROM clause allows outer joins and natural joins to be specified as well.

- Types: NATURAL JOIN, FULL OUTER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN
  - ⇨The keyword "outer" can be omitted for outer joins.

Example: Return all departments (even those without projects) and their projects.

```
SELECT dname, pname
FROM   dept LEFT OUTER JOIN proj ON dept.dno = proj.dno
```

```
SELECT dname, pname
FROM   dept LEFT OUTER JOIN proj USING (dno)
```

```
SELECT dname, pname
FROM   dept NATURAL LEFT JOIN proj
```





## Subqueries in *FROM* Clause

Subqueries are used in the FROM clause to produce temporary table results for use in the current query.

Example: Return the departments that have an employee that makes more than $40,000.

```
SELECT dname
FROM   Dept D, (SELECT ename, dno FROM Emp
                WHERE salary > 40000) E
WHERE D.dno = E.dno
```

- Note: The alias for the derived table is required.





## SQL Querying with Subqueries

***Question:*** What query below is equivalent to:

```
SELECT ename
FROM   emp as E
WHERE  salary > ALL (SELECT salary FROM emp)
```

**A)**

```
SELECT ename
FROM   emp as E
WHERE  salary > (SELECT MAX(salary) FROM emp)
```

**B)**

```
SELECT ename
FROM   emp as E
WHERE  salary > (SELECT SUM(salary) FROM emp)
```





## Subquery Practice Questions

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

1) List all departments that have at least one project.

2) List the employees who are not working on any project.

3) List the employees with title 'EE' that make more than all employees with title 'PR'.

4) Find all employees who work on some project that 'J. Doe' works on.





## SQL Functions

Databases have many built-in functions that can be used when writing queries. Syntax and support varies between systems.

- Date: DATEDIFF, YEAR, GETDATE
- String: CONCAT, UPPER, LEFT, SUBSTRING
- Logical: CASE, IIF, ISNULL
- Aggregate: SUM, COUNT, AVG
- Note: Case-insensitive function names.

Example:

```
SELECT eno, UPPER(ename),
  CASE title WHEN 'EE' THEN 'Engineer'
  WHEN 'SA' THEN 'Admin' ELSE 'Other' END as role,
  year(bdate) as birthYear
FROM emp
WHERE salary * 2 > 60000
```





## SQL Query Summary

The general form of the SELECT statement is:

**SELECT <attribute list>**
**FROM <table list>**
**[WHERE (*condition*)]**
**[GROUP BY <grouping attributes>]**
**[HAVING <group condition>]**
**[ORDER BY <attribute list>]**
**[LIMIT <num> [OFFSET <offset>] ]**

- Clauses in square brackets ([,]) are optional.
- There are often numerous ways to express the same query in SQL.

## Your Own Practice Questions

Relational database schema:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

Given the above schema answer your own English questions using SQL.

Are there some questions that you cannot answer?





## Conclusion

**SQL** is the standard language for querying relational databases.

The **`SELECT`** statement is used to query data and combines the relational algebra operations of selection, projection, and join into one statement

- There are often many ways to specify the same query.
- Aggregate functions, recursive queries, outer joins

**`INSERT`**, **`DELETE`**, and **`UPDATE`** statements are used to modify the data stored in a single relation.





## Objectives

- Be able to write an English query in SQL.
- Be able to convert basic SQL queries into relational algebra expressions.
- Be able to write the various types of queries:
  - basic queries involving selection, projection, and join
  - queries involving renaming and aliasing including queries involving multiple copies of the same relation
  - queries containing aggregate functions and calculated fields
  - nested subqueries and the use of the appropriate operators:
    - comparison operators for single value subqueries
    - `IN`, `NOT IN`, `ANY`, `ALL` for table result subqueries
    - `EXISTS` and `NOT EXISTS` for multiple result subqueries which may or may not contain results

# COSC 304
# Introduction to Database Systems

## Database Design

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Database Design

The ability to design databases and associated applications is critical to the success of the modern enterprise.

Database design requires understanding both the operational and business requirements of an organization as well as the ability to model and realize those requirements in a database.

Developing database and information systems is performed using a ***development lifecycle***, which consists of a series of steps.





## The Importance of Database Design

Just as proper design is critical for developing large applications, success of database projects is determined by the effectiveness of database design.

Some statistics on software projects:

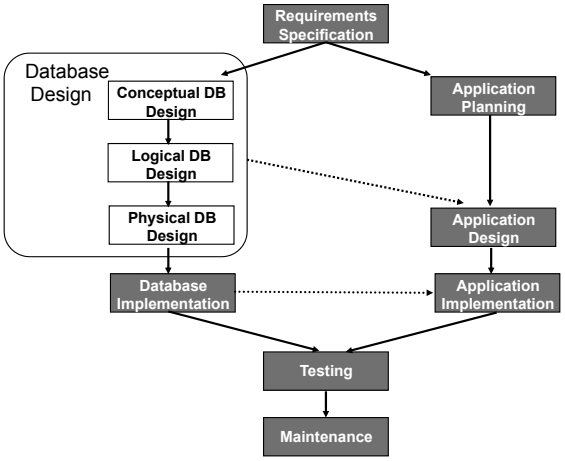
- 80 - 90% do not meet their performance goals
- 80% delivered late and over budget
- 40% fail or abandoned
- 10 - 20% meet all their criteria for success
- Have you been on a project that failed? A) Yes B) No

The primary reasons for failure are improper requirements specifications, development methodologies, and design techniques.





## Database Design Stages








## Specification

One of the primary reasons for a project to fail is that it is not clear what the project should accomplish.

***Specification*** involves *detailed* documentation of:

- understanding how the project fits into the organization
- project goals and success outcomes
- project timelines and deliverables
- measurement criteria for determining project success
- information on users and user requirements

Organizational planning is an early stage of specification where enterprise needs and goals are identified, new technology is evaluated, and new development projects are considered.





## Specification Mission Statements

The ***mission statement*** of a database project is a specification of the major aims and objectives of the project.

- Each mission objective should be stated clearly and preferably have defined metrics to evaluate if it is successfully completed.

The IT manager should construct the mission statement and objectives after consulting with management and users.

- The mission statement may also include:
  - ⇨ the "Project Champion"
  - ⇨ the estimated costs and how these costs will be paid
  - ⇨ standards for database/application development that must be followed
  - ⇨ documentation of privacy and security concerns
  - ⇨ legal issues including copyright when dealing with outside developers
  - ⇨ information on how the project will interface with other systems

A consultant is often asked to bid on a project given its mission statement and must carefully read any legal concerns.

## Specification
## Mission Statement Example

NASA's mission statement when going to the moon:

"I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to Earth."

- (John F. Kennedy May 25, 1961)

NASA fulfilled that goal on July 20, 1969, when Apollo 11's lunar module *Eagle* touched down in the Sea of Tranquility, with Neil Armstrong and Buzz Aldrin aboard. A dozen men would walk on the moon before the Apollo program ended. The last of those men, Gene Cernan, left the desolate lunar surface with these words: "We leave as we came and, God willing, as we shall return, with peace and hope for all mankind."





## Specification
## The "Project Champion"

The "**Project Champion**" is a manager or senior IT person who is the project's promoter and backer.

Many projects fail because no one takes ownership of them.

- Consequently, they take too long, go over budget, and are never deployed effectively.
- When hiring outside consultants, make sure somebody in the organization is the Project Champion.
- For internal projects, a Project Champion is especially important as there are always conflicts over money, developer time, and political issues on making users work on new applications.

**Bottom line:** If no one is willing to be the champion for a project, it is likely that project will not achieve its goals.





## Specification
## Requirements Gathering

Requirements gathering collects details on organizational processes and user issues.

- Often the organization itself does not know this information, and it can only be determined by collecting it from user interviews.
- Through user interviews identify:
  - ⇨Who are the users? Group them into classes.
  - ⇨What do the users do now? (existing systems/processes)
  - ⇨What are the complaints and possibilities for improvement?
- Determine the data used by the organization, identify data relationships, and determine how data is used and generated.
  - ⇨Identify unique fields (keys)
  - ⇨Determine data dependencies, relationships, and constraints (high-level)
  - ⇨Estimate the data sizes and their growth rates





## Specification
## Requirements Gathering (2)

The best way to determine user requirements is to:

**ASK THEM, THEN LISTEN**

The biggest mistake that you can make is to walk into a company and say that:

*"I am going to build you a C# application using Microsoft SQL Server. Give me the data that you have, and I will go away and build you a great system!"*

RULE #1: The user (and managers) are your ultimate consumers; listen and respect their opinions and needs.

RULE #1a: Users (and managers) do not always know what they want or may want the wrong things. Sometimes you must convince them of other alternatives or provide solutions that are good compromises.





## Database Design

The requirements gathering and specification provides you with a high-level understanding of the organization, its data, and the processes that you must model in the database.

Database design involves constructing a suitable model of this information.

Since the design process is complicated, especially for large databases, database design is divided into three phases:

- Conceptual database design
- Logical database design
- Physical database design





## Database Design
## Conceptual Database Design

***Conceptual database design*** involves modeling the collected information at a high-level of abstraction without using a particular data model or DBMS.

High-level modeling languages such as the Entity-Relationship (ER) model and UML are used.

Conceptual database design is *top-down design* as you start by specifying entities (real-world objects) then build up the model by defining new entities, attributes, and relationships.

- ⇨We will also see a bottom-up design technique called *normalization* where you define the attributes first and then use dependency information to group them into relations.

# Database Design
# Logical Database Design

***Logical database design*** is the process of constructing a model of the information in the domain using a particular data model, but independent of the DBMS.

The conceptual model is transformed into a logical model such as the relational model.

- It may also be transformed to other logical models such as the object-oriented model, graphs, JSON, or XML.

Since logical design selects a data model, it is now possible to model the information using the features of that model.

- For example, modeling using the relational model allows you to specify keys, domains, and foreign key constraints.





# Database Design
# Physical Database Design

***Physical database design*** is the process of constructing a physical model of information on the secondary storage in a given data model for a particular DBMS.

Physical database design involves the selection of a database system and determining how to represent the logical model on that DBMS. For the relational model this includes:

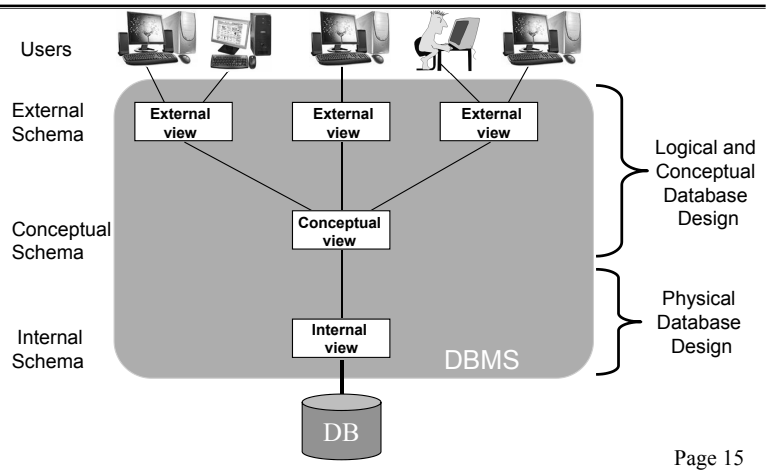
- creating a set of relational tables using the logical model
- increasing performance using indexes
- implementing constraints and security restrictions
- data partitioning and distribution

Physical database design is *how*, and logical database design is the *what*.





# Database Design and the ANSI/SPARC Architecture








# Physical Database Design
# Selecting a DBMS

The process of a selecting a DBMS is not always simple if the organization does not already have one or there are multiple DBMSs that can be selected.

There are several factors to consider when selecting a DBMS:

- **Features**: Does the DBMS have what you need?
  - ⇨Security, constraints, indexes, triggers, SQL-compliance, JDBC/ODBC
- **Compatibility**: Does the DBMS run on your hardware/software?
  - ⇨Can the DBMS interface with existing systems?
- **Performance**: Does the DBMS handle the data/query load?
  - ⇨**Scalability**: Will the DBMS scale if the data/query load grows?
- **Price**: Does the DBMS fit the project budget?
  - ⇨Free is not necessarily cheap. How will the DBMS be supported?





# Application Design

The application can be designed *partially* in parallel with the database design.

During the requirements gathering phase, you can collect information on how the data will be manipulated, the general layout of the screens, and requirements that the application will need from the DBMS.

Two main areas of application design:

- **Transactions** - What queries/updates will the application perform?
- **User interface** - How will the data be displayed on the screen and how can the user initiate transactions?





# Application Design Recommendations

1) Design the logical database schema first before seriously starting the application design.

- ⇨Changes to the logical schema have big impacts on your application and will reduce your development efficiency.

2) Prototype your application, but not your database design.

- ⇨It will reduce your development time if you spend the time getting your database design right before building your application.
- ⇨Flush out as many details as possible before finalizing your DB design and implementing on the DBMS. Then implement the logical design.
- ⇨Prototype your application in stages to get user feedback on the forms and reports as these will frequently change.

3) Isolate database access code into a few classes/methods.

## Implementation

The *implementation* stage builds the database and application designs.

The database designs are implemented in the relational model using SQL DDL.

The application is designed in a programming language such as HTML/JavaScript, PHP, C/C++, or Java.





## Testing and Maintenance

*Testing* and *maintenance* are the two final stages:

- ***Testing*** is the process of executing the application programs with the intent of finding errors.
- ***Operational maintenance*** is the process of monitoring and maintaining the system following installation.

There is a significant cost associated with testing and maintaining existing systems. Good design should allow for a system to accommodate changes with minimal reprogramming.

Maintenance also includes monitoring the DBMS for performance, security violations, and upgrades.





## CASE Tools

Computer-Aided Software Engineering (CASE) tools can make the development easier.

There are design tools that:

- Automatically convert a high-level design (UML diagram) entered graphically into SQL DDL.
- Examine queries for performance and suggest improvements.
- Provide sophisticated application development environments.

Many DBMSs have monitoring and maintenance systems that assist the DBA in detecting performance issues.

- Called "tuning" a database.
  - ⇨ An active area of research is the development of systems that tune themselves!





## Database People DA and DBA

We have seen these two database people before:

- ***Database administrator*** (***DBA***) - responsible for installing, maintaining, and configuring the DBMS software.
- ***Data administrator*** (***DA***) - responsible for organizational policies on data creation, security, and planning.

The DA is involved in the early phases of design including planning the project and conceptual and logical design.

The DBA performs the physical design and actively manages deployed, production systems.

Another common position is a (data) ***architect*** that selects systems to use, makes design decisions, and evaluates current and future systems at the architectural level.





## Conclusion

Database design is critical to the success of database development projects.

Database design is divided into three phases:

- Conceptual database design
- Logical database design
- Physical database design

Effective requirements gathering is an important skill to master.

Projects should have a well-defined mission statement and project champion to increase their probability of success.





## Objectives

- Draw and describe the database development lifecycle.
- Describe the three different steps in database design including the results of each step.
- Describe how the roles of DBA and DA fit into database design. What do these people do?
- List some of the factors to consider when selecting a DBMS for a project.

# COSC 304
# Introduction to Database Systems

## Entity-Relationship Modeling

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Conceptual Database Design

***Conceptual database design*** involves modeling the collected information at a high-level of abstraction without using a particular data model or DBMS.

Since conceptual database design occurs independently from a particular DBMS or data model, we need high-level modeling languages to perform conceptual design.

The entity-relationship (ER) model was originally proposed by Peter Chen in 1976 for conceptual design. We will perform ER modeling using Unified Modeling Language (UML) syntax.





## Entity-Relationship Modeling

***Entity-relationship modeling*** is a top-down approach to database design that models the data as entities, attributes, and relationships.

The ER model refines entities and relationships by including properties of entities and relationships called *attributes*, and by defining *constraints* on entities, relationships, and attributes.

The ER model conveys knowledge at a high-level (conceptual level) which is suitable for interaction with technical and non-technical users.

Since the ER model is data model independent, it can later be converted into the desired logical model (e.g. relational model).





## Example Relation Instances

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

WorksOn Relation

| eno | pno | resp | hours |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |

Proj Relation

| pno | pname | budget | dno |
|---|---|---|---|
| P1 | Instruments | 150000 | D1 |
| P2 | DB Develop | 135000 | D2 |
| P3 | Budget | 250000 | D3 |
| P4 | Maintenance | 310000 | D2 |
| P5 | CAD/CAM | 500000 | D2 |

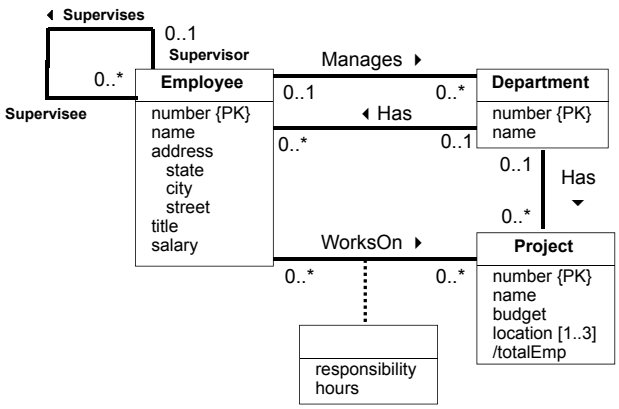
Dept Relation

| dno | dname | mgreno |
|---|---|---|
| D1 | Management | E8 |
| D2 | Consulting | E7 |
| D3 | Accounting | E5 |
| D4 | Development | null |





## ER Model Example in UML notation








## Entity Types

An ***entity type*** is a group of objects with the same properties which are identified as having an independent existence.

- An entity type is the basic concept of the ER model and represents a group of real-world objects that have properties.
  - ⇨Note that an entity type does not always have to be a physical real-world object such as a person or department, it can be an abstract concept such as a project or job.

An ***entity instance*** is a particular example or occurrence of an entity type.

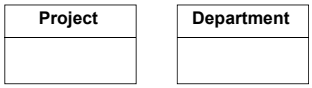
- For example, an entity type is Employee. A entity instance is 'E1 - John Doe'.

An ***entity set*** is a set of entity instances.

# Representing Entity Types

Entity types are represented by rectangles with the name of the entity type in the rectangle.

Examples:




- An entity type name is normally a singular noun.
  - ⇨ That is, use Person instead of People, Project instead of Projects, etc.
- The first letter of each word in the entity name is capitalized.





# Entities Question

***Question:*** How many of the following statements are ***true***?

**1)** Entity types are represented using a rectangle box.
**2)** An entity is always a physical object.
**3)** An entity type is named using a plural noun.
**4)** Employee number is an entity.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





# Relationship Types

A ***relationship type*** is a set of associations among entity types. Each relationship type has a name that describes its function.

A ***relationship instance*** is a particular occurrence of a relationship type that relates entity instances.

- For example, WorksOn is a relationship type. A relationship instance is that 'E1' works on project 'P1' or (E1,P1).

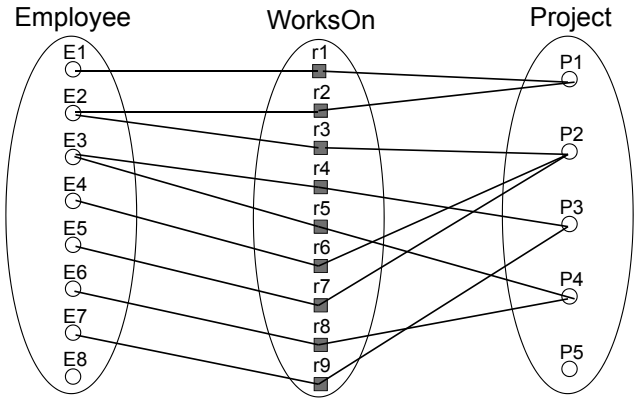
A ***relationship set*** is a set of relationship instances.

There can be more than one relationship between two entity types.





# Visualizing Relationships



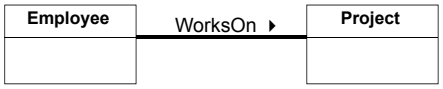

Note: This is an example of a many-to-many relationship. A project can have more than one employee, and an employee can work on more than one project.





# Representing Relationship Types

The relationship type is represented as a labeled edge between the two entity types. The label is applied only in one direction so an arrow indicates the correct way to read it.




- A relationship type name is normally a verb or verb phrase.
- The first letter of each word in the name is capitalized.
- **Do not put arrows on either end of the line.**





# Relationship Degree

The ***degree of a relationship type*** is the number of entity types participating in the relationship.

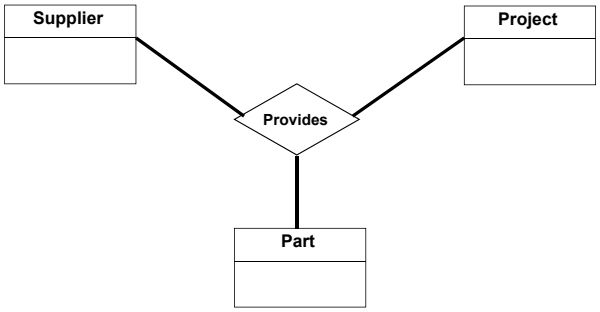
- For example, WorksOn is a relationship type of degree two as the two participating entity types are Employee and Project.
  - ⇨ Note: This is not the same as degree of a relation which was the number of attributes in a relation.

Relationships of degree two are *binary*, of degree three are *ternary*, and of degree four are *quaternary*.

- Relationships of arbitrary degree N are called *n*-ary.

Use a diamond to represent relationships of degree higher than two.

## Ternary Relationship Type Example




A project may require a part from multiple different suppliers.



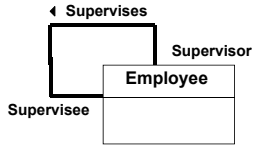


## Recursive Relationships

A ***recursive relationship*** is a relationship type where the same entity type participates more than once in different roles.

- For example, an employee has a supervisor. The supervisor is also an employee. Each *role* has a *role name*.

Example:




- Note that the degree of a recursive relationship is two as the same entity type participates twice in the relationship.
  - ⇨ It is possible for an entity type to be in a relationship more than twice.





## Relationship Question

***Question:*** How many of the following statements are ***true***?

**1)** Relationships are represented using a directed edge (with arrows).

**2)** A relationship is typically named using a verb.

**3)** It is not possible to have a relationship of degree 1.

**4)** The degree of a relationship is the number of attributes it has.

**5)** A diamond is used to represent a relationship of degree larger than 2.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Attributes

An ***attribute*** is a property of an entity or a relationship type.

- For example, entity type Employee has attributes name, salary, title, etc.

Some rules:

- By convention, attribute names begin with a lower case letter.
- Each attribute has a *domain* which is the set of allowable values for the attribute.
- Different attributes may share the same domain, but a single attribute may have only one domain.





## Simple and Complex Attributes

An attribute is a ***simple attribute*** if it contains a single component with an independent existence.

- For example, salary is a simple attribute.
- Simple attributes are often called *atomic* attributes.

An attribute is a ***composite attribute*** if it consists of multiple components each with an independent existence.

- For example, address is a complex attribute because it consists of street, city, and state components (subattributes).

Question: Is the name attribute of Employee simple or complex?





## Single- and Multi-Valued Attributes

An attribute is a ***single-valued attribute*** if it consists of a single value for each entity instance.

- For example, salary is a single-valued attribute.

An attribute is a ***multi-valued attribute*** if it may have multiple values for a single entity instance.

- For example, a telephone number attribute for a person may be multivalued as people may have different phone numbers (home phone number, cell phone number, etc.)

A ***derived attribute*** is an attribute whose value is calculated from other attributes but is not physically stored.

- The calculation may involve attributes within the entity type of the derived attribute and attributes in other entity types.

## Attribute Question

***Question:*** How many of the following statements are ***true***?

**1)** Attributes are properties of either entities or relationships.
**2)** An attribute may be multi-valued.
**3)** A composite attribute contains two or more components.
**4)** Each attribute has a domain representing its data type.
**5)** A derived attribute is physically stored in the database.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Keys

A ***candidate key*** is a minimal set of attributes that uniquely identifies each instance of an entity type.

- For example, the number attribute uniquely identifies an Employee and is a candidate key for the Employee entity type.

A ***primary key*** is a candidate key that is selected to identify each instance of an entity type.

- The primary key is chosen from a set of candidate keys. For instance, an employee may also have SSN as an attribute. The primary key may be either SSN or number as both are candidate keys.

A ***composite key*** is a key that consists of two or more attributes.

- For example, a course is uniquely identified only by the department code (COSC) and the course number within the department (304).





## Key Question

***Question:*** How many of the following statements are ***true***?

**1)** It is possible to have two candidate keys with different numbers of attributes.
**2)** A composite key has more than 1 attribute.
**3)** The computer picks the primary key used in the design.
**4)** A relationship has a primary key.
**5)** An attribute has a primary key.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Representing Attributes

In UML attributes are listed in the rectangle for their entity. Tags are used to denote any special features of the attributes.

- primary key: {PK}, partial primary key: {PPK}, alternate key: {AK}
- derived attribute: */attributeName* (e.g. /totalEmp)
- multi-valued attribute: *attributeName* [*minVals*..*maxVals*]
  - ⇨e.g. phoneNumber [1..3]





## Attributes on Relationships

An attribute may be associated with a relationship type.

For example, the WorksOn relationship type has two attributes: responsibility and hours.

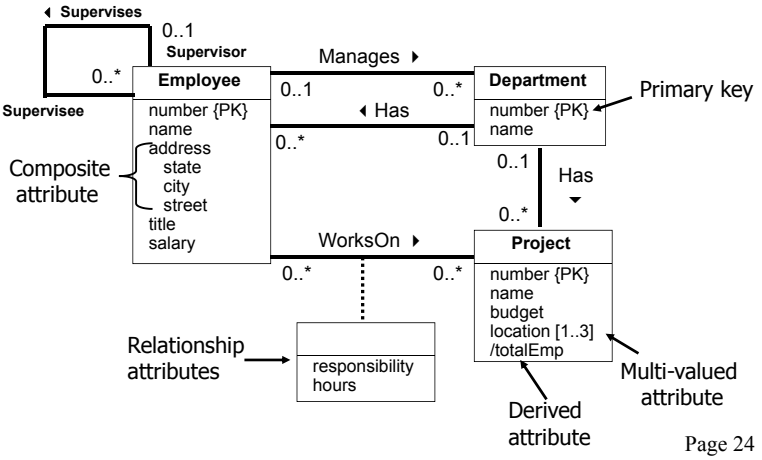
Note that these two attributes belong to the relationship and cannot belong to either of the two entities individually (as they would not exist without the relationship).

Relationship attributes are represented as a separate box connected to the relationship using a dotted line.





## Attributes in UML Notation



# ER Design Question #1

Construct a university database where:

- Each student has an id, name, sex, birth date, and GPA.
- Each professor has a name and is in a department.
- Each department offers courses and has professors. A department has a name and a building location.
- Each course has a name and number and may have multiple sections.
- Each section is taught by a professor and has a section number.
- Students enroll in sections of courses. They may only enroll in a course once (and in a single section). A student receives a grade for each of their course sections.





# Relationship Cardinalities

***Relationship cardinalities*** or ***multiplicities*** are used to restrict how entity types participate in relationships in order to model real-world constraints.

The ***multiplicity*** is the number of possible occurrences of an entity type that may relate to a single occurrence of an associated entity type through a particular relationship.

For binary relationships, there are three common types:

- one-to-one (1:1)
- one-to-many (1:* or 1:N)
- many-to-many (*:* or N:M)





# One-to-One Relationships

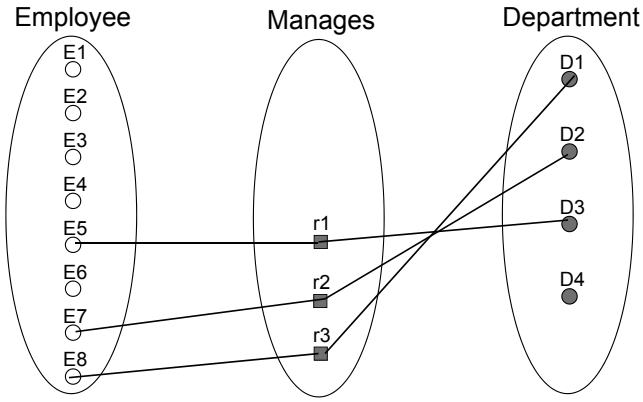
In a one-to-one relationship, each instance of an entity class E1 can be associated with **at most one** instance of another entity class E2 and vice versa.

Example: A department may have only one manager, and a manager may manage only one department.






# One-to-One Relationship Example




Relationship explanation: A department may have only one manager. A manager (employee) may manage only one department.

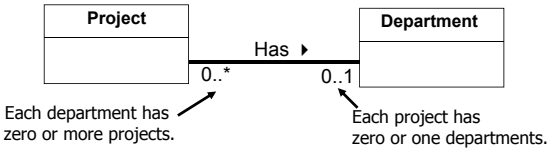




# One-to-Many Relationships

In a one-to-many relationship, each instance of an entity class E1 can be associated with **more than one** instance of another entity class E2. However, E2 can only be associated with **at most one** instance of entity class E1.

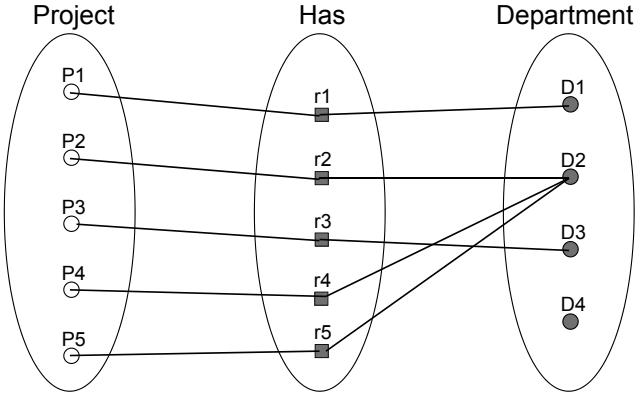
Example: A department may have multiple projects, but a project may have only one department.








# One-to-Many Relationship Example

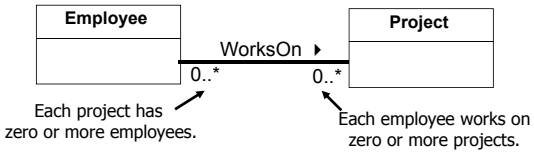



Relationship explanation: A project may be associated with at most one department. A department may have multiple projects.

# Many-to-Many Relationships

In a many-to-many relationship, each instance of an entity class E1 can be associated with **more than one** instance of another entity class E2 and vice versa.

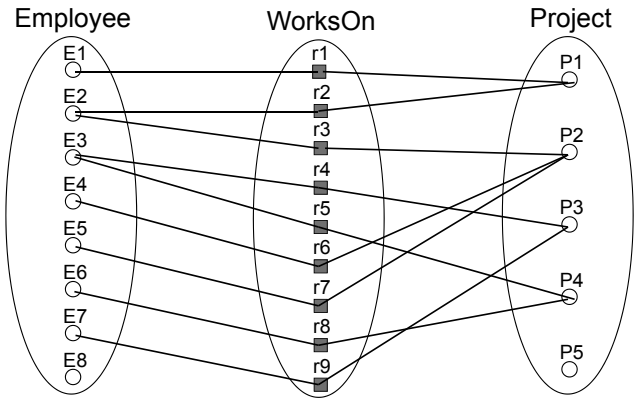
Example: An employee may work on multiple projects, and a project may have multiple employees working on it.








# Many-to-Many Relationship Example








# Participation Constraints

***Cardinality*** is the *maximum* number of relationship instances for an entity participating in a relationship type.

***Participation*** is the *minimum* number of relationship instances for an entity participating in a relationship type.

- Participation can be *optional* (zero) or *mandatory (1 or more)*.

If an entity's participation in a relationship is mandatory (also called *total* participation), then the entity's existence depends on the relationship.

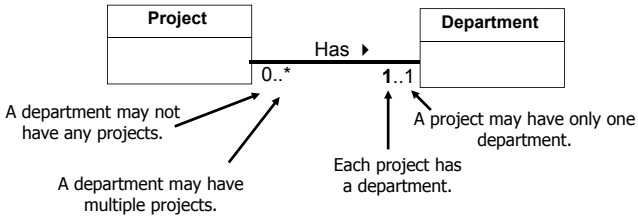
- Called an *existence dependency*.





# Participation Constraints Example

Example: A project is associated with one department, and a department may have zero or more projects.

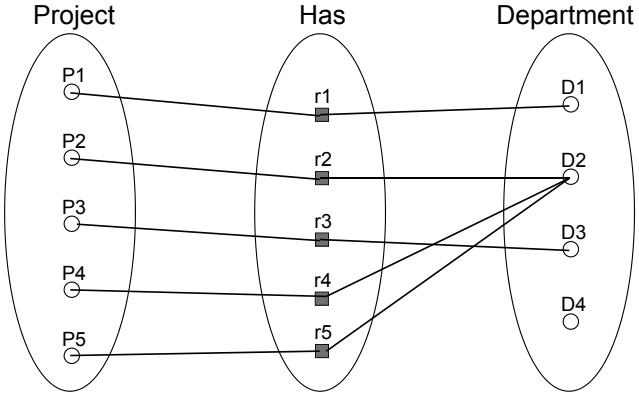



Note: Every project must participate in the relationship (mandatory).





# One-to-Many Participation Relationship Example



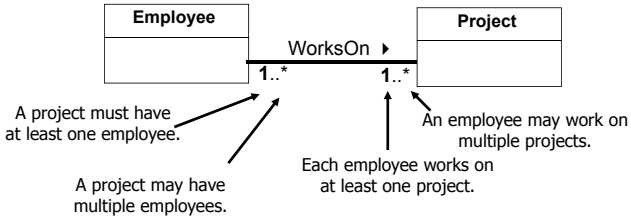

Relationship explanation: A project must be associated with one department. A department may have zero or more projects.



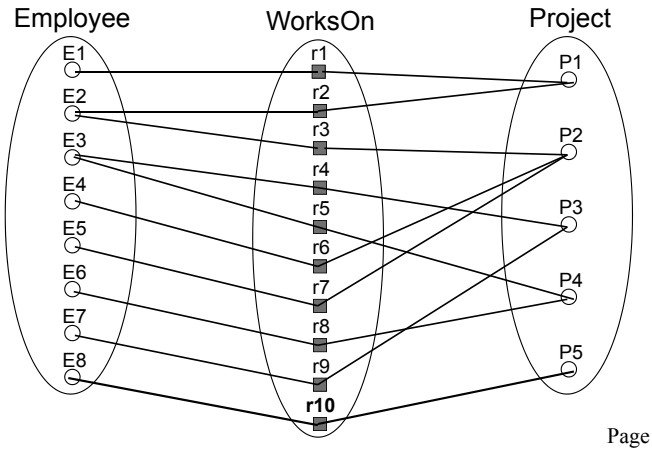


# Participation Constraints Example 2

Example: A project must have one or more employees, and an employee must work on one or more projects.



# Many-to-Many Relationship Participation Example



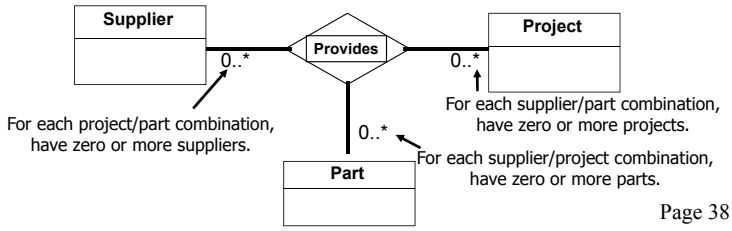





# Multiplicity of Non-Binary Relationships

The multiplicity in a complex relationship of an entity type is the number of possible occurrences of that entity-type in the *n*-ary relationship when the other (*n*-1) values are fixed.

Example: A supplier may provide zero or more parts to a project. A project may have zero or more suppliers, and each supplier may provide to the project zero or more parts.








# Challenges with Multiplicity of Non-Binary Relationships

The participation constraint for *N*-ary relationships (minimum cardinality) is ***ambiguous*** in UML notation. Example:

- Constraint: A project has at least 1 supplier per part.
- Initial idea: Multiplicity 1..* beside Supplier should force their to be a Supplier for each Part/Project combination.
- Problem: Two different interpretations of Part/Project combination results in unforeseen consequences:
  - ⇨Actual tuple: All entities must always participate (as have actual tuples) so minimum values for all entities would be 1.
  - ⇨Potential tuple: 1 beside Supplier implies always a Supplier for every Part/Project combination. Not true in practice.

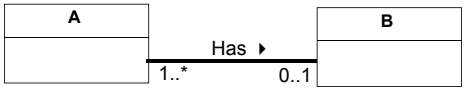
Bottom line: We will avoid problem of specifying participation constraints for *N*-ary relationships. One way to avoid it is convert a relationship into an entity with *N* binary relationships.





# Relationship Cardinality Question

***Question:*** How many of the following statements are ***true***?




**1)** An entity of type A must be related to an entity of type B.

**2)** An entity of type A may be related to more than one entity B.

**3)** This is a 1-to-many relationship between A and B.

**4)** An entity of type B must be related to an entity of type A.

**5)** An entity of type B must be related to more than one entity of type A.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





# Multiplicity Practice Question

Consider the university database developed before. Write multiplicities into the ER diagram given that:

- A department must offer at least 2 courses and no more than 20 courses. Courses are offered by only one department.
- A course may have multiple sections, but always has at least one section.
- A student may enroll for courses (but does not have to).
- A professor may be in multiple departments (at least 1), and a department must have at least 3 professors.
- A section is taught by at least one professor, but may be taught by more than one. A professor does not have to teach.





# Strong and Weak Entity Types

A ***strong entity type*** is an entity type whose existence is not dependent on another entity type.

- A strong entity type always has a primary key of its own attributes that uniquely identifies its instances.

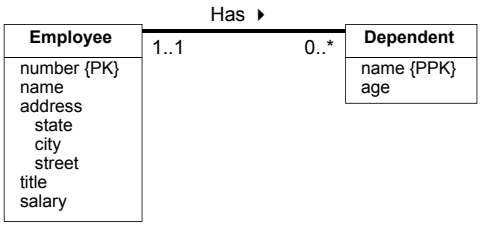
A ***weak entity type*** is an entity type whose existence is dependent on another entity type.

- A weak entity type does not have a set of its own attributes that uniquely identifies its instances.

A common example of strong and weak entity types are employees and their dependents:

- An employee is a strong entity because it has an employee number to identify its instances.
- A dependent (child) is a weak entity because the database does not store a key for each child, but rather they are identified by the parent's employee number and their name.

## Weak Entities in UML








## Strong and Weak Entity Question

***Question:*** How many of the following statements are ***true***?

**1)** A weak entity has its own primary key.

**2)** A strong entity has its own primary key.

**3)** A weak entity must be associated (identified) by a strong entity.

**4)** A weak entity can have a relationship with another entity besides its identifying strong entity.

**5)** The attribute(s) of a weak entity used to identify it with its associated strong entity are noted as `{PPK}` in the UML model.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Problems with ER Models

When modeling a Universe of Discourse (UoD) using ER models, there are several challenges that you have.

The first, basic challenge is knowing when to model a concept as an entity, a relationship, or an attribute.

In general:

- Entities are nouns.
  - ⇨You should be able to identify a set of key attributes for an entity.
- Attributes are properties and may be nouns or adjectives.
  - ⇨Use an attribute if it relates to one entity and does not have its own key.
  - ⇨Use an entity if the concept may be shared by entities and has a key.
- Relationships should generally be binary.
  - ⇨Note that non-binary relationships can be modeled as an entity instead.





## Modeling Traps

There are several different "modeling traps" that you can fall into when designing your ER model.

Two connection traps that we will look at are:

- Fan traps
- Chasm traps



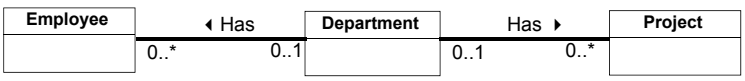


## Fan Traps

A ***fan trap*** is when a model represents a relationship between entity types, but the pathway between certain entity instances is ambiguous.

- Often occurs when two or more one-to-many relationships fan out (come from) the same entity type.

Example: A department has multiple employees, a department has multiple projects, and each project has multiple employees.

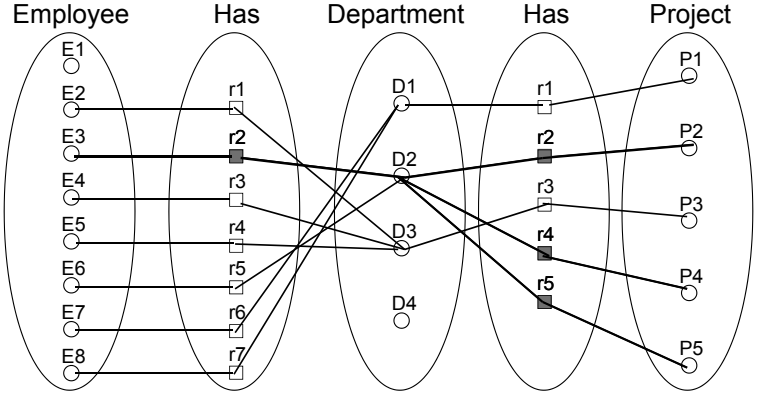



Now answer the question, which projects does employee E3 work on?





## Fan Trap Example



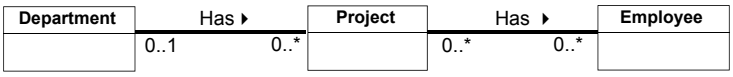

Which projects does employee E3 work on?

# Chasm Traps

A ***chasm trap*** occurs when a model suggests that a relationship between entity types should be present, but the relationship does not actually exist. (*missing* relationship)

- May occur when there is a path of optional relationships between entities.

Example: A department has multiple employees, a department has multiple projects, and each project has multiple employees.

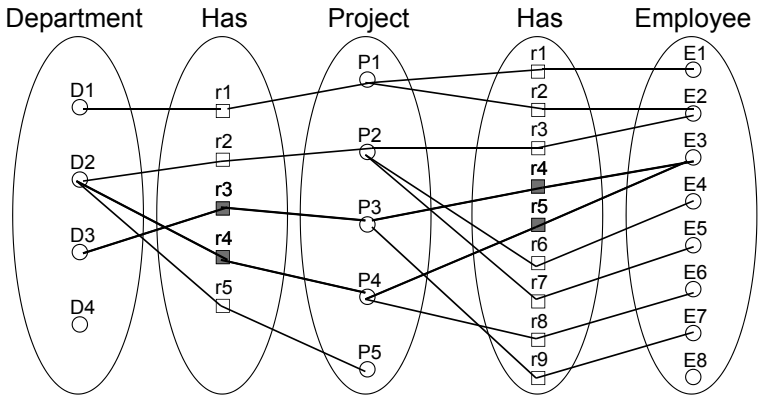



Now answer the question, what department is employee E3 in?





# Chasm Trap Example




Which department is employee E3 in?
What are the employees of department D2?





# Good Design Practices

When designing ER models, there are several things that you should consider:

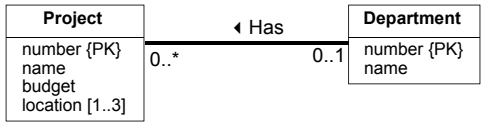
- 1) Avoid redundancy - do not store the same fact more than once in the model.
- 2) Do not use an entity when you can use an attribute instead.
- 3) Limit the use of weak entity sets.



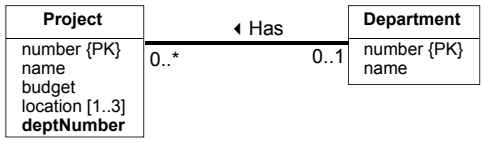


# Good Design Practices Avoiding Redundancy Example

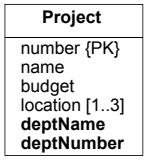
Good:




Wrong:




Bad:







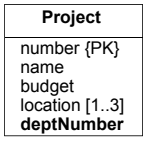

# Good Design Practices Entity versus Attribute Example

An entity should only be used if one of these conditions is true:

- 1) The entity set should contain at least one non-key attribute.
- 2) It is the many side in a many-to-one relationship.

Example: Projects have a department. A department only has a number.

Acceptable:




Most general:






# Good Design Practices Weak Entity Sets

Avoid the use of weak entity sets in modeling. Although at first glance there are very few natural keys based on the properties of objects (name, age, etc.), there are many good human-made keys that can be used (SSN, student#, etc.)

Whenever possible use a global, human-made key instead of a weak entity. Note that sometimes a weak entity is required if no such global key exists.

- For example, a database storing the students of multiple universities could not use a single student# as a key as it is unlikely that universities could agree on a global numbering system. Rather, student becomes a weak entity with partial key student# and identifying entity the university the student attends.

## Simplifications

Two common relationship simplifications:

- Many-to-many relationship ⇨Two one-to-many relationships
- Higher order relationships ⇨ binary relationships

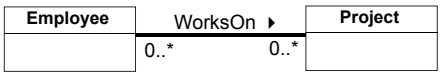




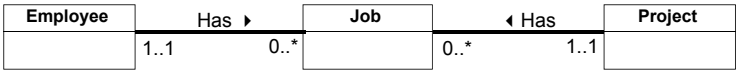
## Many-to-Many Relationship Simplification

A many-to-many relationship can be converted into one entity with two 1:N relationships between the new entity and the original entities participating in the relationship.

Original:




Simplified:

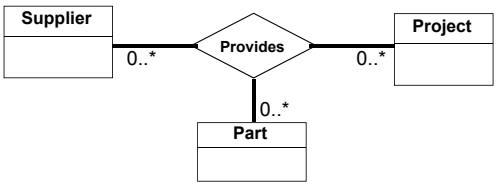




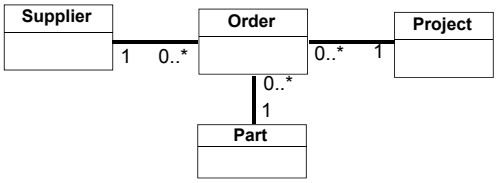



## Higher Degree Relationships Simplified using a Weak Entity

Original:




With weak entity:








## ER Design Example TPC-H Standard Schema

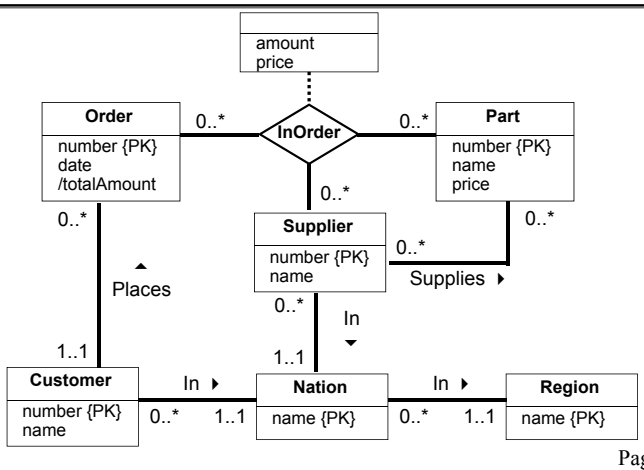
The database will store order information:

- Each order has a numeric key, a customer, a date, a total order amount, and a list of parts.
- A part in an order has an amount and a price paid and is supplied by a certain supplier.
- Each part has a numeric key, a name, and a price and may be supplied by multiple suppliers.
- A supplier has a key and a name and may supply multiple parts.
- A customer has a key and a name.
- Each supplier and customer is located in a nation.
- Each nation is located in a region (continent).





## ER Design Example TPC-H Standard Schema Answer








## ER Design Question #2

Construct a fish store database where:

- A fish store maintains a number of aquaria (tanks), each with a number, name, volume and color.
- Each tank contains a number of fish, each with an id, name, color, and weight.
- Each fish is of a particular species, which has a id, name, and preferred food.
- Each individual fish has a number of events in its life, involving a date and a note relating to the event.

## ER Design Question #3

Construct an invoice database where:

- An invoice is written by a sales representative for a single customer and has a unique ID. An invoice has a date and total amount and is comprised of multiple detail lines, containing a product, price and quantity.
- Each sales representative has a name and can write many invoices, but any invoice is written by a single representative.
- Each customer has a unique id, name, and address and can request many invoices.
- Products have descriptions and weights and are supplied by vendors. Each product has a unique name for a particular vendor. A product is supplied by only one vendor.
- A vendor has an id and an address.





## Conclusion

Conceptual design is performed at a high-level of abstraction involving entities, relationships, and attributes.

- An entity type is a group of entities with the same properties.
  - ⇨Entities may be strong (have unique key) or weak (no unique key).
- A relationship type is an association between entities.
  - ⇨A relationship may involve two or more entities and may be recursive.
  - ⇨A relationship has two types of constraints:
    - Participation - minimum # of times an entity must be involved in relationship
    - Cardinality - maximum # of times an entity can be involved in relationship
  - ⇨Common relationship multiplicities are: 1:1, 1:*, *:*.
- Attributes are properties of entities or relationships.

The ER model using UML syntax is used for database design.

It is possible to fall into design traps, so practice is necessary to become a good conceptual designer.





## Objectives

- Explain the relationship between conceptual design and ER modeling/UML.
- Be able to define: entity type, relationship type, degree of a relationship, recursive relationship, attribute, attribute domain, simple attribute, composite attribute, single-valued attribute, multi-valued attribute, derived attribute
- Be able to define: candidate key, primary key, composite key
- Be able to explain the difference between a strong entity type and a weak entity type.
- Be able to explain multiplicity and participation and how they are used in modeling.
- Be able to describe fan traps and chasm traps.

Be able to model a domain explained in an English paragraph in an ER diagram using UML notation.

# COSC 304
# Introduction to Database Systems
# Enhanced Entity-Relationship (EER) Modeling

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Enhanced Entity-Relationship Modeling

***Enhanced Entity-Relationship (EER) modeling*** is an extension of ER modeling to include object-oriented concepts such as:

- superclasses and subclasses
- specialization and generalization
- aggregation and composition

These modeling constructs may allow more precise modeling of systems that are object-oriented in nature such as:

- CAD/CAM systems (Computer-Aided Design/Manufacturing)
- GIS (Geographical Information Systems)





## Review: Superclasses and Subclasses

The object-oriented ideas of inheritance and superclasses and subclasses are taught during programming in an OO language such as Java.

A ***superclass*** is a general class that is extended by one or more subclasses.

A ***subclass*** is a more specific class that extends a superclass by inheriting its methods and attributes and then adding its own methods and attributes.

***Inheritance*** is the process of a subclass inheriting all the methods and attributes of a superclass.



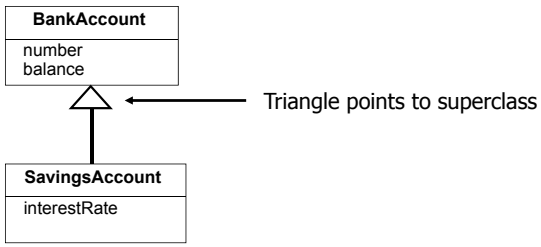


## Superclasses and Subclasses Example

Java code:

```
public class SavingsAccount extends BankAccount
```

UML class diagram:

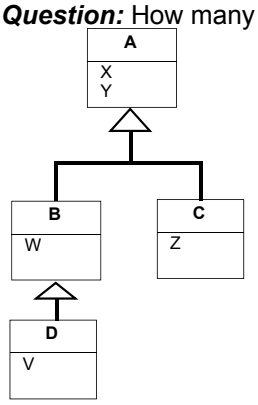







## Superclasses and Subclasses Question

***Question:*** How many of the following statements are ***true***?




**1)** D is a superclass.
**2)** D has 1 attribute.
**3)** B and C are subclasses of A.
**4)** B inherits V from D.
**5)** D inherits attribute X.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## When to use EER Modeling?

It is important to emphasize that many database projects do not need the object-oriented modeling features of EER modeling.

Remember the goal of conceptual modeling is to produce a model that is simple and easy to understand.

Do not introduce complicated subclass/superclass relationships if they are not needed.

Only use the EER modeling constructs if they offer a *significant advantage* over regular ER modeling.

## When to use EER Modeling? (2)

EER modeling is especially useful when the domain being modeled is object-oriented in nature and the use of inheritance reduces the complexity of the design.

There are two common cases where EER modeling is useful instead of basic ER modeling:

- 1) When using *attribute inheritance* can reduce the use of nulls in a single entity relation (that contains multiple subclasses).
- 2) Subclasses can be used to explicitly model and name subsets of entity types that participate in their own relationships.





## When to use EER Modeling? Using Attribute Inheritance

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

Note that the `title` attribute indicates what job the employee does at the company. Consider if each job title had its own unique information that we would want to record such as:

- `EE`, `PR` - programming language used (`lang`), DB used (`db`)
- `SA`, `ME` - MBA? (`MBA`), `bonus`





## When to use EER Modeling? Using Attribute Inheritance (2)

We could represent all these attributes in a single relation:

| eno | ename | bdate | title | salary | supereno | dno | lang | db | MBA | bonus |
|---|---|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | | C++ | MySQL | | |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | | | N | 2000 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | | | N | 3000 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Java | Oracle | | |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | | | Y | 4000 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | C++ | DB2 | | |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | | | N | 3000 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | | D1 | | | Y | 6000 |

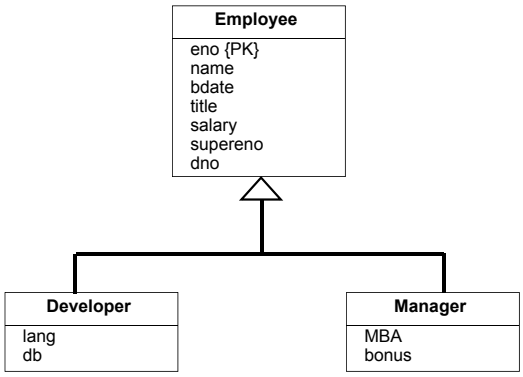
Note the wasted space as attributes that do not apply to a particular subclass are `NULL`.





## When to use EER Modeling? Using Attribute Inheritance (3)

A better solution would be to make two subclasses of `Employee` called `Developer` and `Manager`:








## When to use EER Modeling? Using Attribute Inheritance (4)

Resulting relations:

Employee Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

Developer Relation

| eno | lang | db |
|---|---|---|
| E1 | C++ | MySQL |
| E4 | Java | Oracle |
| E6 | C++ | DB2 |

Manager Relation

| eno | MBA | bonus |
|---|---|---|
| E2 | N | 2000 |
| E3 | N | 3000 |
| E5 | Y | 4000 |
| E7 | N | 3000 |
| E8 | Y | 6000 |





## Generalization and Specialization

Subclasses and superclasses are created by using either generalization or specialization.

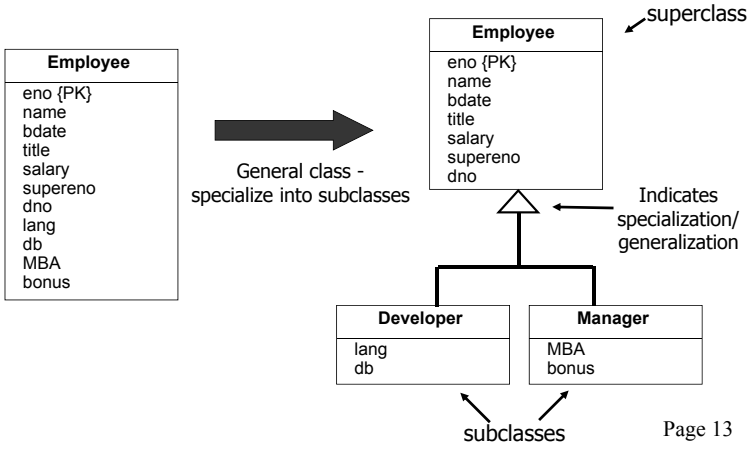
***Specialization*** is the process of creating more specialized subclasses of an existing superclass.

- Top-down process: Start with a general class and then subdivide it into more specialized classes.
  - ⇨ The specialized classes may contain their own attributes. Attributes common to all subclasses remain in the superclass.

***Generalization*** is the process of creating a more general superclass from existing subclasses.

- Bottom-up process: Start with specialized classes and try to determine a general class that contains the attributes common to all of them.

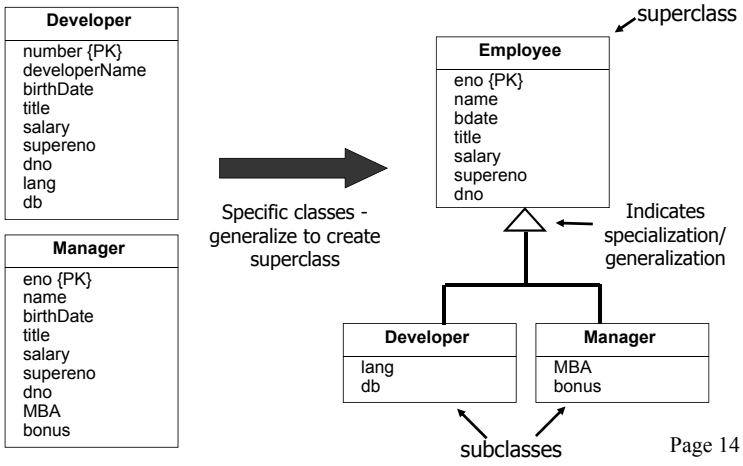
# Specialization Example








# Generalization Example








# Constraints on Generalization and Specialization

There are two types of constraints associated with generalization and specialization:

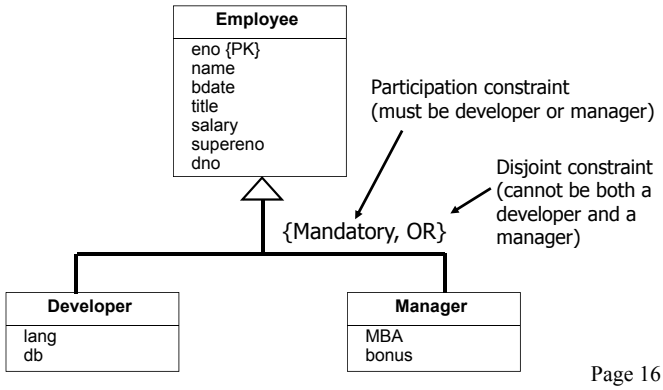
- ***Participation constraint*** - determines if every member in a superclass must participate as a member of one of its subclasses.
  - ⇨It may be `optional` for a superclass member to be a member of one of its subclasses, or it may be `mandatory` that a superclass member be a member of one of its subclasses.
- ***Disjoint constraint*** - determines if a member of a superclass can be a member of one or more than one of its subclasses.
  - ⇨If a superclass object may be a member of only one of its subclasses this is denoted by `OR` (subclasses are *disjoint*).
  - ⇨Otherwise, `AND` is used to indicate that it may be in more than one of its subclasses.





# Constraints Example

An employee must be either a developer or a manager, but cannot be both.

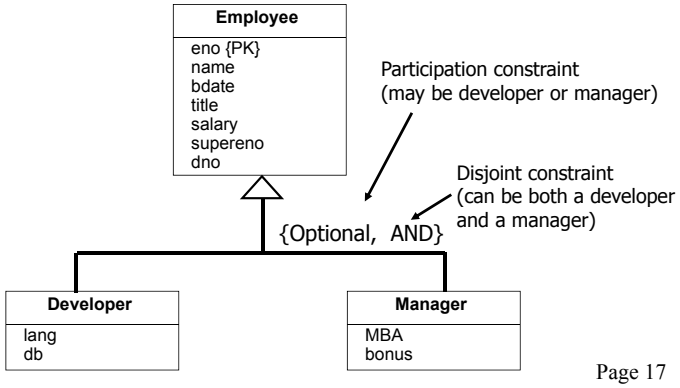







# Constraints Example (2)

An employee may specialize as a developer or manager. An employee may be both a manager and developer.








# General Predicate Constraints

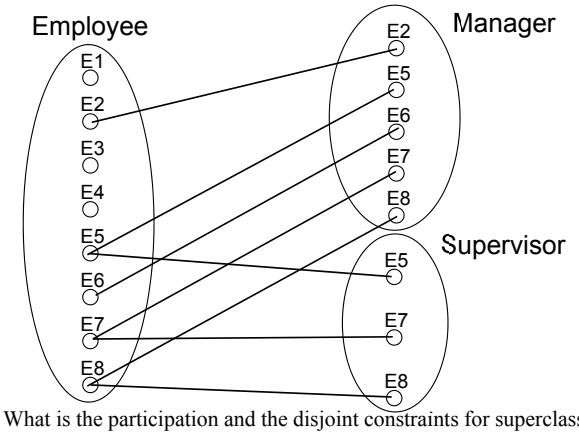
***Predicate-defined constraints*** specify when an object participates in a subclass using a certain rule.

- For example, a subclass called `RichEmployees` can be defined with a membership predicate such as `salary >100000`.

***Attribute-defined subclasses*** are a particular type of predicate-defined constraint where the value of an attribute(s) determines if an object is a member of a subclass.

- For example, the `title` field could be used as a *defining attribute* for the `Developer` and `Manager` subclasses.
  - ⇨`Emp` is in `Developer` if `title = 'EE' or 'PR'`
  - ⇨`Emp` is in `Manager` if `title = 'ME' or 'SA'`

## Constraints Question




Note: What is the participation and the disjoint constraints for superclass `Employee` (with subclasses `Manager` and `Supervisor`) given these instances?





## Relationship Constraints vs. Inheritance Constraints

There is a parallel between relationship constraints on associations/relationships and inheritance constraints on superclasses and subclasses.

- Minimum # of occurrences – called participation constraint in both cases
- Maximum # of occurrences – called cardinality constraint for relationships and disjoint constraint for subclasses

Possible combinations:

| Subclass Constraints | Relationship Constraints |
|---|---|
| Optional, AND | 0..* |
| Optional, OR | 0..1 |
| Mandatory, AND | 1..* |
| Mandatory, OR | 1..1 |





## EER Question

***Question:*** How many of the following statements are ***true***?

**1)** Generalization is a bottom-up process.

**2)** In an UML diagram, the inheritance arrow points towards the superclass.

**3)** OPTIONAL and MANDATORY are possible choices for the participation constraint.

**4)** If the disjoint constraint is AND, a given object can be a member of multiple subclasses.

**5)** If the participation constraint is OPTIONAL, the disjoint constraint must be AND.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Multiple Inheritance

If each class only has one superclass, then the class diagram is said to be a ***specialization*** or ***type hierarchy***.

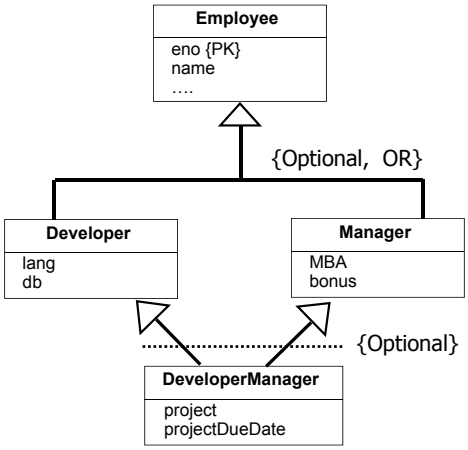
If a class may have more than one superclass, then the class diagram is said to be a ***specialization*** or ***type lattice***.

Although multiple inheritance is powerful, it should be avoided if possible.





## Multiple Inheritance Example



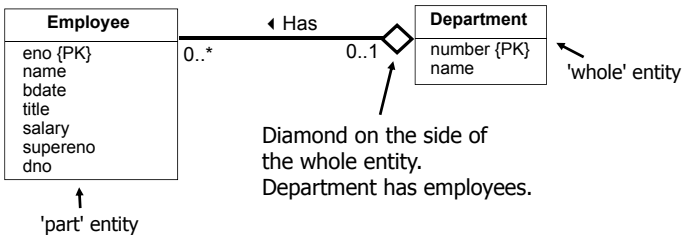





## Aggregation

***Aggregation*** represents a `'HAS-A'` or `'IS-PART-OF'` relationship between entity types. One entity type is the *whole*, the other is the *part*.

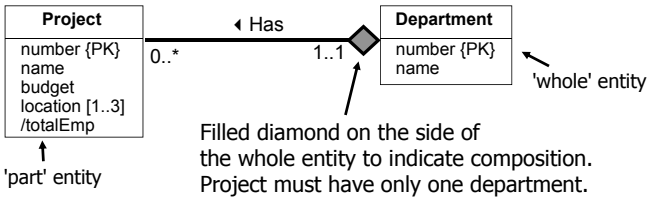
Example:



# Composition

***Composition*** is a stronger form of aggregation where the part cannot exist without its containing whole entity type and the part can only be part of one entity type.

Example:

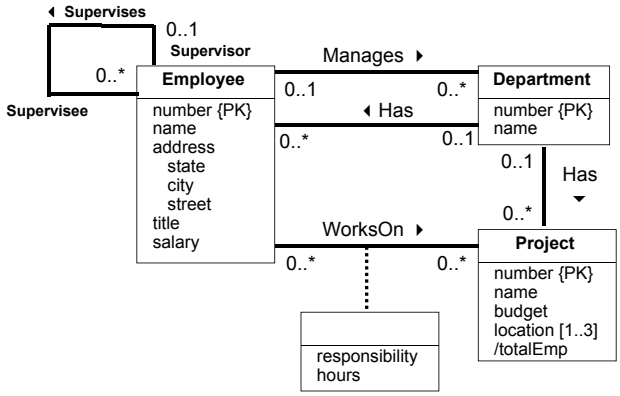



Note: The min-max constraint on the whole side of the relationship (in this case department) must always be 1..1 when modeling composition. Why?





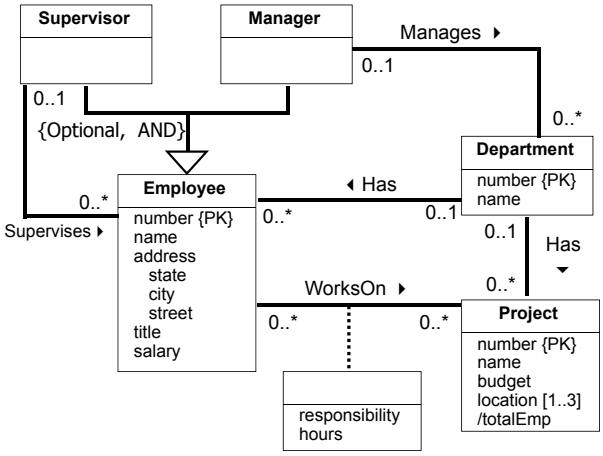
# Original ER Model Example








# EER Model Example








# Conclusion

The ***Enhanced Entity-Relationship*** model (***EER***) model allows for object-oriented design features to be captured.

***Generalization*** and ***specialization*** are two complementary processes for constructing superclasses and subclasses.

***Participation*** and ***disjoint constraints*** apply to subclasses.

- Participation of a superclass may be `mandatory` or `optional` in a subclass.
- A superclass may only be a member of one subclass (disjoint constraint indicated by `OR`) or multiple (indicated by `AND`).

Aggregation and composition are used to model `HAS-A` or `PART-OF` relationships.

The features of EER modeling are rarely needed in most database design projects.





# Objectives

Given an EER diagram, recognize the subclasses, superclasses, and constraints using the notation.

Explain the difference between the participation constraint and the disjoint constraint.

Explain the difference between aggregation and composition.

Given an EER diagram, list the attributes of each class including attributes inherited from superclasses.

# COSC 304
# Introduction to Database Systems

## ER to Relational Mapping

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## ER Model to Relational Schemas

These notes will describe how to convert an ER diagram (or EER diagram) into a corresponding relational schema.

Conceptual database design produces a conceptual ER model. This conceptual model is then converted into the relational model (which is a logical model).

Note that although it is possible to design using the relational model directly, it is normally more beneficial to perform conceptual design using the ER model first.





## ER Model to Relational Schemas (2)

Converting an ER model to a relational database schema involves 7 steps.

In general, these steps convert entities to relations and ER relationships to relations. For 1:1 and 1:N relationships, foreign keys can be used instead of separate relations.

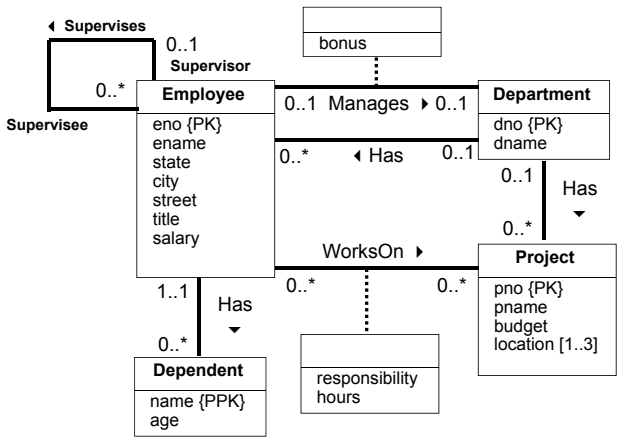
Handling subclasses and superclasses in the EER model requires an extra conversion step.

After conversion is performed, normalization and optimization are often performed to improve the relational schema.





## ER Model Example

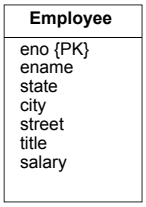







## ER to Relational Mapping
## Step #1: Convert Strong Entities

Step #1: Convert each strong entity to a relation.




Employee (eno, ename, state, city, street, title, salary)

- Notes:
  - 1) Attributes of the entity type become attributes of the relation.
  - 2) Include only simple attributes in relation. For composite attributes, only create attributes in the relation for their simple components.
  - 3) Multi-valued attributes are handled separately (in step #6).
  - 4) The primary key of the relation is the key attributes for the entity.





## ER to Relational Mapping
## Current Relational Schema - Step #1

Employee (eno, ename, state, city, street, title, salary)

Project (pno, pname, budget)

Department (dno, dname)

# ER to Relational Mapping
# Step #2: Convert Weak Entities

Step #2: Convert each weak entity into a relation with foreign keys to its identifying relations (entities).

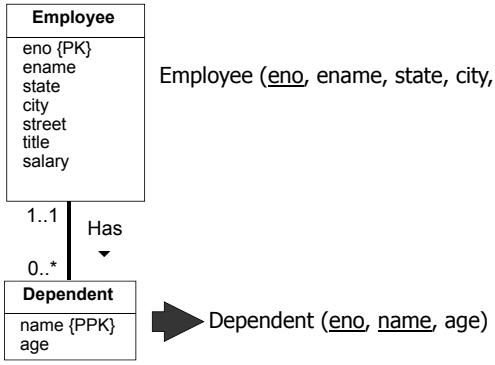
For each weak entity *W* with identifying owners $E_1, E_2, \ldots, E_n$ create a relation *R*:

- Identify relations $R_1, R_2, \ldots, R_n$ for entity types $E_1, E_2, \ldots, E_n$.
- The primary key of *R* consists of the primary keys of $R_1, R_2, \ldots, R_n$ plus the partial key of the weak entity.
- Create a foreign key in *R* to the primary key of each relation $R_1, R_2, \ldots, R_n$.
- Attributes are converted the same as strong entities.





# ER to Relational Mapping
# Step #2: Convert Weak Entities (2)




Employee (<u>eno</u>, ename, state, city, street, title, salary)

Dependent (<u>eno</u>, <u>name</u>, age)





# ER to Relational Mapping
# Current Relational Schema - Step #2

**Dependent (<u>eno</u>, <u>name</u>, age)**

Employee (<u>eno</u>, ename, state, city, street, title, salary)

Project (<u>pno</u>, pname, budget)

Department (<u>dno</u>, dname)





# ER to Relational Mapping
# Steps #3-5: Convert Relationships

Steps 3 to 5 convert *binary* relationships of cardinality:

- 1:1 - Step #3
- 1:N - Step #4
- M:N - Step #5

Note that M:N relationships are the most general case, and the conversion algorithm for these relationships can be applied to 1:1 and 1:N as well.

- However, for performance reasons, it is normally more efficient to perform different conversions for each relationship type.
- In general, each ER relationship can be mapped to a relation. However, for 1:1 and 1:N relationships, it is more efficient to combine the relationship with an existing relation instead of creating a new one.

Relationships that are not binary are handled in step #7.





# ER to Relational Mapping
# Step #3: Convert 1:1 Relationships

Step #3: Convert binary 1:1 relationships into a `UNIQUE` foreign key reference from one relation to the other.

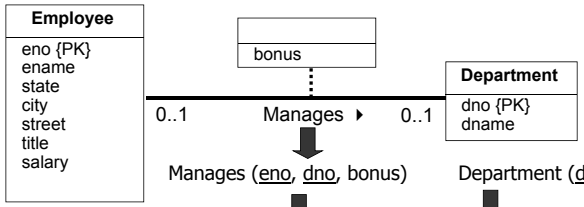
Given a binary 1:1 relationship *R* between two entities $E_i$ and $E_j$:

- Identify the corresponding relations $R_i$ and $R_j$.
- Chose one of the relations, say $R_i$, and:
  - ⇨ Add the attributes of *R* to $R_i$.
  - ⇨ Add the primary key attributes of $R_j$ to $R_i$, and create a foreign key reference to $R_j$ from $R_i$.
  - ⇨ Declare these primary key attributes of $R_j$ to be `UNIQUE`.
- Notes:
  - ⇨ You can select either $R_i$ or $R_j$. Typically, it is best to select the relation that is guaranteed to always participate in the relationship or the one that will participate the most in the relationship.





# ER to Relational Mapping
# Step #3: Convert 1:1 Relationships (2)




Manages (<u>eno</u>, <u>dno</u>, bonus)

Department (<u>dno</u>, dname)

Employee (<u>eno</u>, ename, state, city, street, title, salary)

Department (<u>dno</u>, dname, **mgreno**, **bonus**)

Note: Renamed `eno` to `mgreno` for clarity.

# ER to Relational Mapping
# Step #4: Convert 1:N Relationships

Step #4: Convert binary 1:N relationships between into a foreign key reference from the N-side relation to the 1-side relation.

Given a binary 1:N relationship *R* between two entities $E_i$ and $E_j$:

- Identify the corresponding relations $R_i$ and $R_j$.
- Let $R_i$ be the N-side of the relation.
  - ⇨ Add the attributes of *R* to $R_i$.
  - ⇨ Add the primary key attributes of $R_j$ to $R_i$, and create a foreign key reference to $R_j$ from $R_i$.

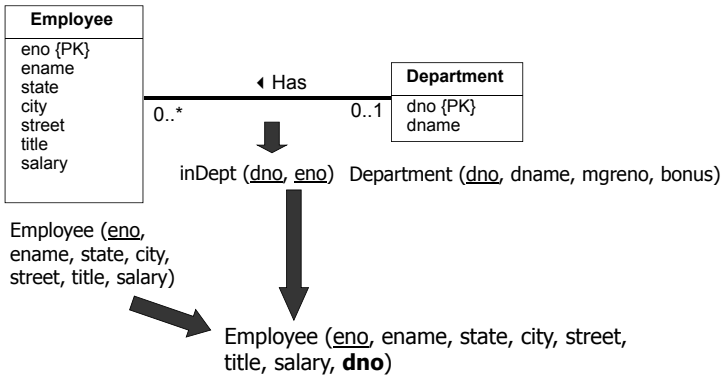
Notes:

- Unlike 1:1 relationships, you must select the N-side of the relationship as the relation containing the foreign key and relationship attributes.





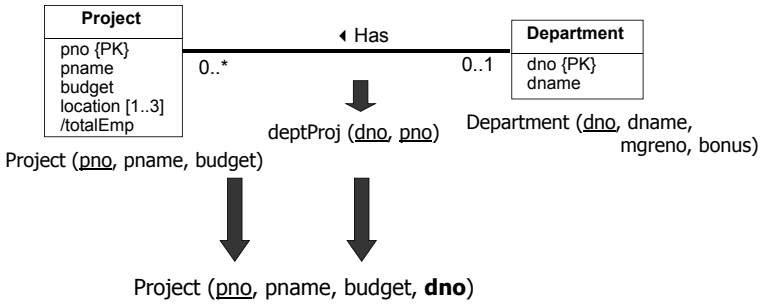
# ER to Relational Mapping
# Step #4: Convert 1:N Relationships







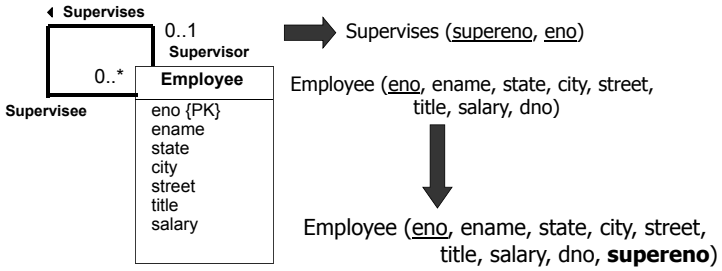

# ER to Relational Mapping
# Step #4: Convert 1:N Relationships







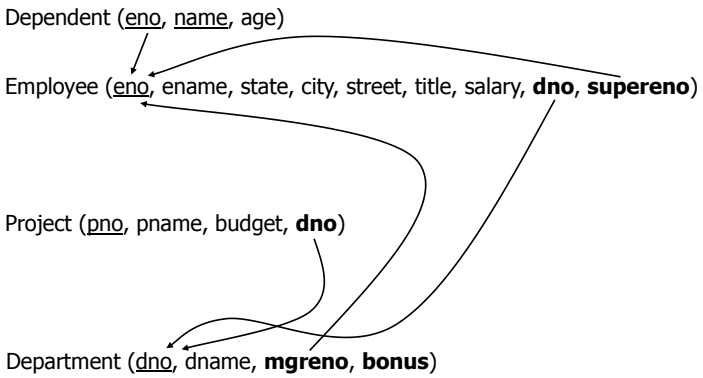

# ER to Relational Mapping
# Step #4: Convert 1:N Relationships








# ER to Relational Mapping
# Current Relational Schema - Step #4







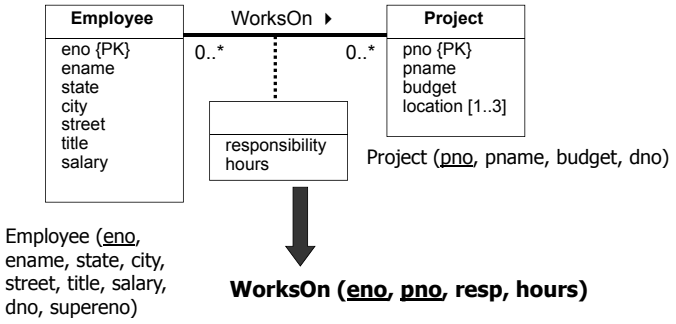

# ER to Relational Mapping
# Step #5: Convert M:N Relationships

Step #5: Convert binary M:N relationships into a new relation with foreign keys to the two participating entities.

Given a binary M:N relationship between entities $E_i$ and $E_j$:

- Identify the corresponding relations $R_i$ and $R_j$.
- Create a new relation *R* representing the relationship where:
  - ⇨ *R* contains the relationship attributes.
  - ⇨ The primary key of *R* is a composite key consisting of the primary keys of $R_i$ and $R_j$.
  - ⇨ Add the primary key attributes of $R_i$ and $R_j$ to *R*, and create a foreign key reference to $R_i$ from *R* and to $R_j$ from *R*.

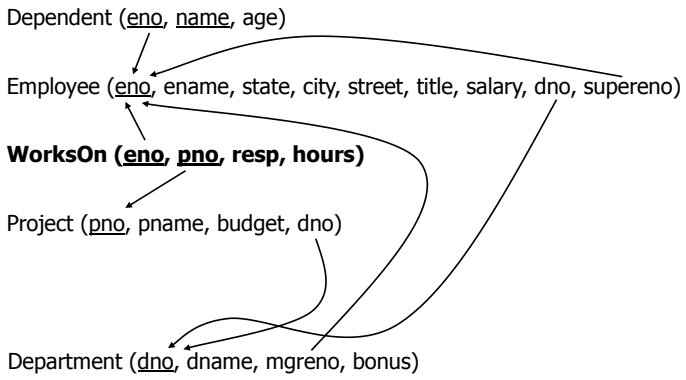
# ER to Relational Mapping
# Step #5: Convert M:N Relationships








# ER to Relational Mapping
# Current Relational Schema - Step #5








# ER to Relational Mapping
# Step #6: Convert Multi-Valued Attributes

Step #6: Convert a multi-valued attribute into a relation with composite primary key consisting of the attribute value plus the primary key of the attribute's entity.

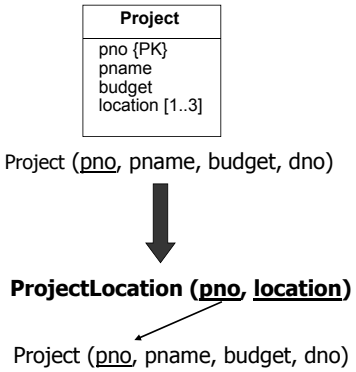
Given a multi-valued attribute *A* of entity $E_i$:

- Identify the corresponding relation $R_i$.
- Create a new relation *R* representing the attribute where:
  - *R* contains the simple, single-valued attribute *A*.
  - Add the primary key attributes of $R_i$ to *R*, and create a foreign key reference to $R_i$ from *R*.
  - The primary key of *R* is a composite key consisting of the primary key of $R_i$ and *A*.





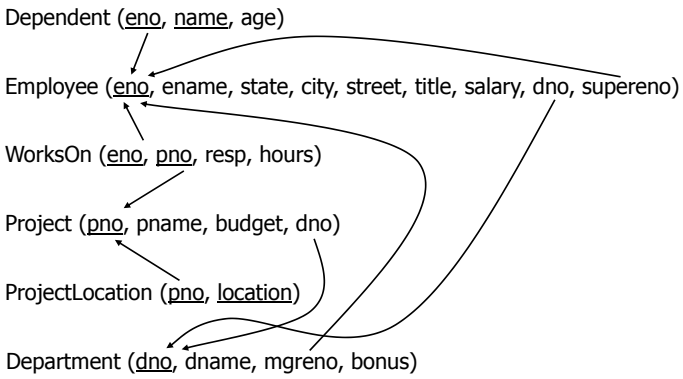
# ER to Relational Mapping
# Step #6: Convert Multi-Valued Attributes








# ER to Relational Mapping
# Final Relational Schema







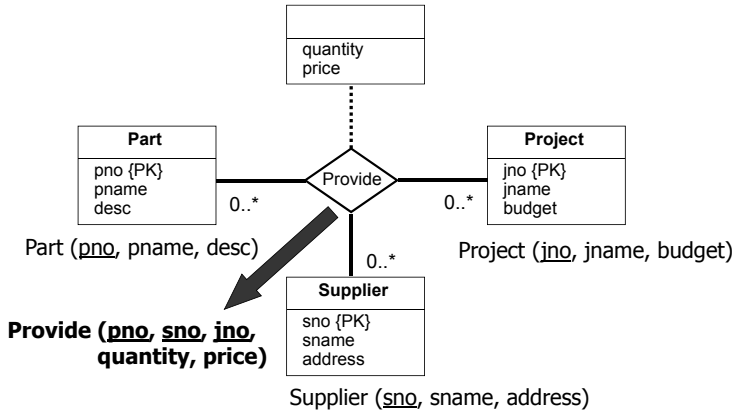

# ER to Relational Mapping
# Step #7: Convert n-ary Relationships

Step #7: Convert *n*-ary relationships by creating a new relation to represent the relationship and creating foreign keys that reference the related entities.

Given an *n*-ary relationship between entities $E_1, E_2, \ldots, E_n$:

- Identify relations $R_1, R_2, \ldots, R_n$ for entity types $E_1, E_2, \ldots, E_n$.
- Create a new relation *R* to represent the relationship.
- The primary key of *R* consists of the primary keys of $R_1, R_2, \ldots, R_n$.
- Create a foreign key in *R* to the primary key of each relation $R_1, R_2, \ldots, R_n$.
- Attributes of the relationship become attributes of *R*.

# ER to Relational Mapping
# Step #7: Convert n-ary Relationships




Part (pno, pname, desc)

Project (jno, jname, budget)

**Provide (pno, sno, jno, quantity, price)**

Supplier (sno, sname, address)





# EER to Relational Mapping

An additional step is necessary to convert subclasses and superclasses to the relational model.

We have several different approaches:

- 1) Create a separate relation for each superclass and subclass.
  - ⇨Most general technique that we will use.
- 2) Create relations for subclass only.
  - ⇨Only works if superclass has mandatory participation.
- 3) Create a single relation with one type attribute.
  - ⇨Attribute is used to indicate the type of object (subclass) in the row.
  - ⇨Works only if the subclasses are disjoint.
- 4) Create a single relation with multiple type attributes.
  - ⇨Have a boolean valued attribute for each subclass. True if in subclass.
  - ⇨Works if subclasses may be overlapping.





# ER to Relational Mapping
# Step #8: Convert Subclasses

Step #8: Convert subclasses and superclasses by creating a relation for each subclass and superclass. Link the subclasses to the superclass using foreign key references.

Given a superclass *C* and set of subclasses $S_1$, $S_2$, …, $S_n$:

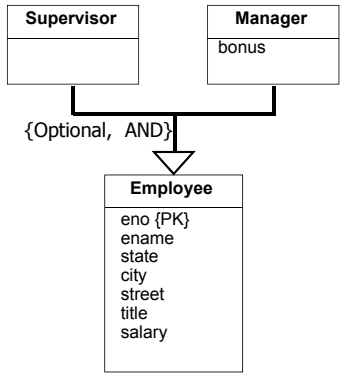
- Create a relation *R* for *C*.
- The primary key for *R* is the primary key of the superclass.
- Create relations $R_1$, $R_2$, …, $R_n$ for subclasses $S_1$, $S_2$, …, $S_n$.
- The primary key for each $R_i$ is the primary key of the superclass.
- For each $R_i$, create a foreign key to *R* using the primary key attributes.





# ER to Relational Mapping
# Step #8: Convert Subclasses

Supervisor (eno)




Manager (eno, bonus)

Employee (eno, ename, state, city, street, title, salary, supereno)





# Summary of
# ER to Relational Mapping

| ER Model | Relational Model |
|---|---|
| Entity Type | Relation |
| 1:1 or 1:N Relationship Type | Foreign key (from N-side to 1-side) |
| M:N Relationship Type | "Relationship" relation and 2 foreign keys |
| *n*-ary Relationship Type | "Relationship" relation and *n* foreign keys |
| Simple attribute | Attribute |
| Composite attribute | Set of simple component attributes |
| Multi-valued attribute | Relation and foreign key |
| Key attribute | Primary key attribute |





# ER to Relational Mapping Question

***Question:*** How many of the following statements are ***true***?

**1)** The M:N relationship mapping rule could be applied to 1:1 and 1:N relationships, as it is more general.

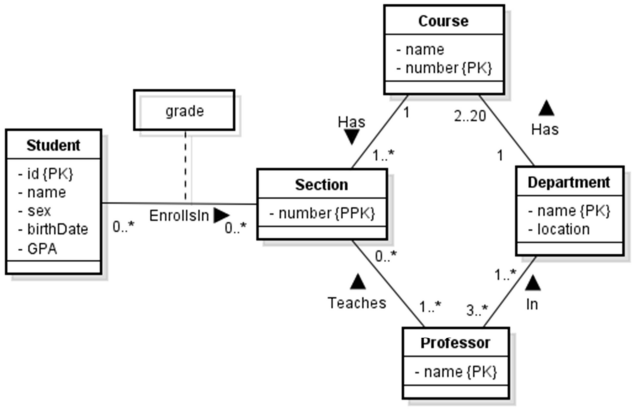
**2)** A weak entity will always have primary key attributes from the identifying entity.

**3)** The designer has a choice on which side to put the foreign key when mapping a 1:N relationship.

**4)** When mapping a multi-value attribute, the new table containing the multi-value attribute will have a composite key.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

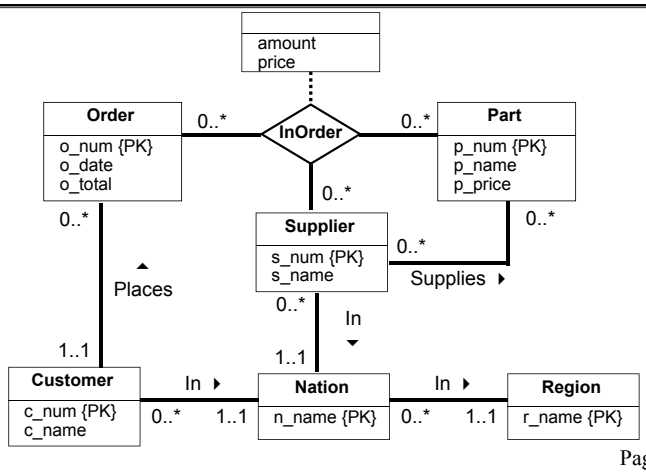
# ER to Relational Mapping University Question








# ER to Relational Mapping TPC-H Standard Question








# Conclusion

There is a straightforward algorithm for converting ER models to relational schemas.

The algorithm involves 7 steps for converting regular ER models, and 8 steps for converting EER models.

In general, these steps convert entities to relations and ER relationships to relations. For 1:1 and 1:N relationships, foreign keys can be used instead of separate relations.





# Objectives

Given an ER/EER diagram, be able to convert it into a relational schema using the seven/eight steps.

Be able to discuss the different ways of converting subclasses/superclasses into relational schemas.

# COSC 304
# Introduction to Database Systems

## Database Programming

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Database Programming Overview

Most user interaction with a database is through programs instead of directly using SQL commands to the database.

A programmer developing a database application executes SQL directly or indirectly in code using one of these methods:

- standard application programming interfaces (APIs) such as JDBC
- custom APIs specific to the database vendor
- object-relational mapping technologies

All methods must process queries in code, send them to the database, retrieve database results, and map the result data into program variables so that they can be used by the program.





## JDBC Overview

**JDBC** is the most popular method for accessing databases using Java programs.

JDBC is an ***application programming interface*** (***API***) that contains methods to connect to a database, execute queries, and retrieve results.

For each DBMS, the vendor writes a JDBC driver that implements the API. Application programs can access different DBMSs simply by changing the driver used in their program.





## JDBC Interfaces

The JDBC API consists of a set of interfaces.

- A Java *interface* is an abstract class consisting of a set of methods with no implementation.
- To create a JDBC driver, the DBMS vendor must implement the interfaces by defining their own classes and writing the code for the methods in the interface.

The main interfaces in the JDBC API are:

- `Driver` - The main class of the entire driver.
- `Connection` - For connecting to the DB using the driver.
- `Statement` - For executing a query using a connection.
- `ResultSet` - For storing and manipulating results returned by a `Statement`.
- `DatabaseMetaData` - For retrieving metadata (schema) information from a database.





## JDBC Program Example

```
import java.sql.*;
```
← Import JDBC API

DB Connection Info

```
public class TestJDBCMySQL
{  public static void main(String[] args)
   {  String url = "jdbc:mysql://cosc304.ok.ubc.ca/WorksOn";
      String uid = "user";
      String pw = "testpw";

      try {         // Load driver class
         Class.forName("com.mysql.jdbc.Driver");
      }
      catch (java.lang.ClassNotFoundException e) {
         System.err.println("ClassNotFoundException: " +e);
      }
```

Load JDBC Driver
(for MySQL – optional)





## JDBC Program Example (2)

Make DB connection

Create statement

Execute statement

Iterate through ResultSet

```
      Connection con = null;
      try {
         con = DriverManager.getConnection(url, uid, pw);
         Statement stmt = con.createStatement();
         ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                            FROM Emp");
         System.out.println("Employee Name,Salary");
         while (rst.next())
            System.out.println(rst.getString("ename")
                  +","+rst.getDouble("salary"));
      }
      catch (SQLException ex) { System.err.println(ex);}
      finally
      {  if (con != null)
            try
            {  con.close(); }
            catch (SQLException ex) { System.err.println(ex); }
      }
   }
}
```

# JDBC Program Example Try-with-Resources Syntax (Java 7+)

```
try (Connection con = DriverManager.getConnection(url, uid, pw);
     Statement stmt = con.createStatement();)
{
   ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                      FROM Emp");
   System.out.println("Employee Name,Salary");
   while (rst.next())
      System.out.println(rst.getString("ename")
                         +","+rst.getDouble("salary"));
}
catch (SQLException ex)
{
   System.err.println(ex);
}
```

Declare resources

`Statement` and `Connection` objects closed by end of `try`





# JDBC `Driver` Interface

The `Driver` interface is the main interface that must be implemented by a DBMS vendor when writing a JDBC driver.

- The class itself does not do very much except allow a connection to be made to a database through the driver.
- Note that you do not call the `Driver` class directly to get a connection. Drivers self-register with the `DriverManager` so `Class.forName()` is no longer needed.
- When you call `DriverManager.getConnection()`, the `DriverManager` will attempt to locate a suitable driver.





# JDBC `Driver` Interface (2)

The `DriverManager` determines which JDBC driver to use based on the URL used as a parameter:

```
// Components of a URL
String url = "jdbc:mysql://cosc304.ok.ubc.ca/testDB";
```

JDBC protocol

subprotocol - used to select driver to use

URL of DB server

Name of DB on server

The interface method `Driver.connect()` is called by the `DriverManager` on the driver object that will allow access based on the subprotocol specified in the URL.

- Inside this method the JDBC driver writer performs the actions to connect to the DBMS and return a `Connection` object.
- A `try .. catch` is also needed here if the driver is unable to successfully connect to the DB at the specified URL.





# JDBC Driver Vendor Compliance

Each DBMS vendor may create a JDBC driver and provide any level of JDBC support that they desire.

- Some methods in the JDBC API are not available on all drivers.
- The basic support required is Entry Level SQL92.

Historical JDBC versions with key features:

- JDBC 1.0 (1997) - `Driver`, `Connection`, `Statement`, `ResultSet`, `DatabaseMetadata`, `ResultSetMetaData`
- JDBC 2.0 (1999) - BLOBs, updatable `ResultSet`, pooled connections
- JDBC 3.0 (2001) - improved transactions (savepoints), SQL99, retrieve auto-generated keys, update BLOBs
- JDBC 4.0 (2006) - XML support, better BLOB support, auto driver loading, do not need `Class.forName()`
- JDBC 4.1 (Java 7), JDBC 4.2 (Java 8)





# JDBC `Connection` Interface

The `Connection` interface contains abstract methods for managing a connection or session.

- A connection is opened after the call to `getConnection()` and should be explicitly closed when you are done.
- The `Connection` interface is used to create statements for execution on the database.

```
Connection con = DriverManager.getConnection(url, uid, pw);
Statement stmt = con.createStatement();
...
con.close();
```





# `Connection` Interface and MetaData

A `Connection` to a database can also be used to retrieve the database metadata (or schema).

- This is useful when you are writing generic tools where you do not know the schema of the database that you are querying in advance.

The method `getMetaData()` can be used to retrieve a `DatabaseMetaData` object.

## DatabaseMetaData Example

```
String []tblTypes = {"TABLE"};// What table types to retrieve

try {
    DatabaseMetaData dmd = con.getMetaData();  // Get metadata
    ResultSet rs1, rs2, rs5;

    System.out.println("List all tables in database: ");

    rs1 = dmd.getTables(null, null, "%", tblTypes);
    while (rs1.next()) {
        String tblName = rs1.getString(3);

        Statement stmt = con.createStatement();
        rs2 = stmt.executeQuery("SELECT Count(*) FROM "+tblName);
        rs2.next();
        System.out.println("Table: "+tblName+" # records: "+
                           rs2.getInt(1));
```





## DatabaseMetaData Example (2)

```
        rs5 = dmd.getColumns(null,null,tblName,"%");
        System.out.println("  Attributes: ");

        while (rs5.next()) {
            System.out.println(rs5.getString(4));
        }
    } // end outer while
} // end try
```





## JDBC `Statement` Interface

The `Statement` interface contains abstract methods for executing a single static SQL statement and returning the results it produces.

```
Statement stmt = con.createStatement();
ResultSet rst = stmt.executeQuery("SELECT ename, salary
                                   FROM Emp");
```

- The `Statement` object is created by calling `Connection.createStatement()`.
- The statement is then executed by calling `executeQuery()` and passing the SQL string to execute.





## JDBC `Statement` Interface (2)

There are two important variations of executing statements that are important and are used often.

- 1) The `Statement` executed is an `INSERT`, `UPDATE`, or `DELETE` and no results are expected to be returned:

```
rowcount = stmt.executeUpdate("UPDATE Emp Set salary=0");
```

- 2) The `Statement` executed is an `INSERT` which is creating a new record in a table whose primary key field is an autonumber field:

```
rowcount = stmt.executeUpdate("INSERT Product
                               VALUES ('Prod. Name')",
                               Statement.RETURN_GENERATED_KEYS );
ResultSet autoKeys = stmt.getGeneratedKeys();
```





## `PreparedStatement` and `CallableStatement`

There are two special types of `Statement` objects:

- `PreparedStatement` - extends `Statement` and is used to execute precompiled SQL statements.
  - ⇨Useful when executing the same statement multiple times with different parameters as the DBMS can optimize its parsing and execution.
  - ⇨Also useful to prevent SQL injection attacks.

```
String SQL = "UPDATE Emp SET salary = ? WHERE ID = ?";
PreparedStatement pstmt = con.prepareStatement(SQL);
pstmt.setBigDecimal(1, 55657.34); // Set parameters
pstmt.setString(2,"E1");
int rowcount = pstmt.executeUpdate();
```

- `CallableStatement` - extends `PreparedStatement` and is used to execute stored procedures.
  - ⇨Stored procedures are precompiled SQL code stored at the database that take in parameters for their execution.





## JDBC `ResultSet`

The `ResultSet` interface provides methods for manipulating the result returned by the SQL statement.

- Remember that the result is a relation (table) which contains rows and columns.
- The methods provide ways of navigating through the rows and then selecting columns of the current row.
- A `ResultSet` object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row.
- The `next()` method moves the cursor to the next row, and because it returns false when there are no more rows in the `ResultSet` object, it can be used in a while loop to iterate through the result set.

## JDBC `ResultSet` (2)

By default a `ResultSet` is not *updatable* and the cursor can move *forward-only* (can only use `next()` method).

```
while (rst.next())
{   System.out.println(rst.getString("ename")
                              +","+rst.getDouble(2));
}
```

- Remember, the first call to `next()` places the row cursor on the first row as the cursor starts off before the first row.
- Use the `getType()` methods to retrieve a particular type.
  - `getArray(), getBlob(), getBoolean(), getClob(), getDate(), getDouble(), getFloat(), getInt(), getLong(), getObject(), getString(), getTime()`
  - All methods take as a parameter the column index in the `ResultSet` (indexed from 1) or the column name and return the requested type.
  - Java will attempt to perform casting if the type you request is not the type returned by the database.





## Scrollable `ResultSets`

It is also possible to request `ResultSets` that allow you to navigate backwards as well as forwards.

- Request `ResultSet` type during `createStatement`.

```
// rs will be scrollable and read-only
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");
```

Scrollable `ResultSets` allow you to navigate in any direction through it, and these methods can now be used:

- `absolute(int row)` - set cursor to point to the given row (starting at 1)
- `afterLast(), beforeFirst(), first(), last(), next(), previous()`
- Scrollable ResultSets were introduced in JDBC 2.0 API and may be less efficient than forward-only ResultSets.





## Updatable `ResultSets`

*Updatable* `ResultSets` allow you to update fields in the query result and update entire rows.

Updating an existing row:

```
// rs will be scrollable and updatable record set
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");

rs.absolute(2);               // Go to the 2nd row
rs.updateString(2,"Joe Blow");// Change name of employee
rs.updateRow();               // Update data source
```





## Updatable `ResultSets` (2)

Adding a new row to a `ResultSet`:

```
// rs will be scrollable and updatable record set
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT eno, ename FROM Emp");

rs.moveToInsertRow();         // Move cursor to insert row
rs.updateString(1,"E9");
rs.updateString("ename","Joe Smith");
rs.insertRow();               // Insert new row in DB
rs.moveToCurrentRow();        // Move cursor to row you were on
                              //  before insert
```





## JDBC API Question

***Question:*** Which one is not a JDBC API interface?

**A)** `Driver`
**B)** `Connection`
**C)** `Statement`
**D)** `ResultSet`
**E)** `HashMap`





## JDBC API Question

***Question:*** Select a **true** statement.

**A)** When a `ResultSet` is first opened, the current row is 0.
**B)** When a `ResultSet` is first opened, the current row is 1.
**C)** When asking for columns, the first column is index 0.
**D)** The method call `first()` is allowed for a forward-only `ResultSet`.

## Custom APIs

Many databases have custom APIs for various languages.

- For example, MySQL has APIs for C/C++ and PHP in addition to ODBC and JDBC drivers.

When to use a custom API?

- Custom APIs may be useful when the ODBC/JDBC standard does not provide sufficient functionality.
- Also useful if *know* that application will only access one database. (Be careful with this .. think of the future).

Custom APIs may have improved performance and increased functionality. The disadvantage is that your code is written for a specific DBMS which makes changes difficult.

- If you use a custom API, always isolate the database access code to a few general classes and methods!





## Aside: Direct Connection using `Driver`

It is possible to directly create a driver class object and call its connect method instead of using `DriverManager`.

- NOT recommended as puts vendor specific code in your code!

```
// Create new structure for passing info to database
Properties prop = new Properties();
prop.put("user","rlawrenc");
prop.put("password","pw");

Connection con = null;
try {
    Driver d = new com.mysql.jdbc.Driver();
    con = d.connect(url, prop);
```

This also demonstrates the use of the `Properties` class that allows you to pass key/value pairs to the database that are not in the URL. Can also use with `DriverManager`:

```
con = DriverManager.getConnection(url, prop);
```





## Aside: Making a Connection using `DataSource`

`DataSource` in package `java.sqlx` is the preferred way for connecting to a database as it hides the URL and vendor specific code (if used in conjunction with JNDI).

```
import javax.sql.DataSource;
import com.mysql.jdbc.jdbc2.optional.*;

MysqlDataSource ds = new MysqlDataSource();
ds.setUser("rlawrenc");
ds.setPassword("pw");
ds.setServerName("cs-suse-4.ok.ubc.ca");
ds.setDatabaseName("WorksOn");
con = ds.getConnection();
```

Note without Java Naming and Directory Interface (JNDI) that vendor specific code is still in the application.





## Object-Relational Mapping Java Persistence Architecture (JPA)

A huge challenge with database programming is converting the database results to and from Java objects. This is called object-relational mapping, and it is tedious and error-prone.

- ***Impedance mismatch*** - Database returns values in tables and rows and Java code manipulates objects, classes, and methods.

Various vendors (e.g. Hibernate) have produced object-relational mapping technologies that help the programmer convert database results into Java objects.

The Java Persistence Architecture (JPA) has been developed as a standard interface. Vendors can then implement the interface in their products.





## JDBC Question

Create a JDBC program that:

- Connects to `WorksOn` database on `cosc304.ok.ubc.ca`.
- Prints on the console each department and its list of projects.

Variant:

- Output in reverse order by department number. Two versions:
  - ⇨Change SQL
  - ⇨Use scrollable ResultSets (hint: previous() method).

Challenge:

- Improve your code so that it prints the department number, name, and how many projects in that department THEN the list of projects.





## Conclusion

**JDBC** is a standard APIs for connecting to databases using Java. Querying using JDBC has these steps:

- Load a `Driver` for the database
- Make a `Connection`
- Create a `Statement`
- Execute a query to produce a `ResultSet`
- Navigate the `ResultSet` to display results or update the DB

There are other ways for accessing a database:

- Custom APIs specific to each database vendor
- object-relational mapping and JPA

***Java Persistence Architecture*** (***JPA***) is a standard API for object-relational mapping.

# Objectives

- Explain the general steps in querying a database using Java.
- List the main JDBC classes (`Driver`, `Connection`, `Statement`, `ResultSet`) and explain the role of each in a typical program.
- Discuss the different types of `ResultSets` including scrollable and updatable `ResultSets`.
- Write a simple JDBC program (given methods of the JDBC API).
- Discuss and explain the advantages and disadvantages of using a standard API versus a vendor-based APIs.
- Explain object-relational mapping and the impedance mismatch.

# COSC 304
# Introduction to Database Systems

## Database Web Programming

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Database Web Programming Overview

Web applications consist of a client interface and a server component.

The client interface does not access the database directly. The server code accesses the database and provides the data as needed to the client.

- The server code may be JSP/Servlets, PHP, Python, etc.
- The client code is HTML/JavaScript.

HTTP is a stateless protocol which requires special handling to remember client state.





## Static versus Dynamic Content

***Static*** content: the HTML is created once and is always the same (although it may include client-side scripts (JavaScript)).

***Dynamic*** content: the HTML is produced when requested. Requires the web server to dynamically generate the output.





## Dynamic HTML and JDBC

Dynamic HTML involves constructing HTML output dynamically on the server then sending it to the client.

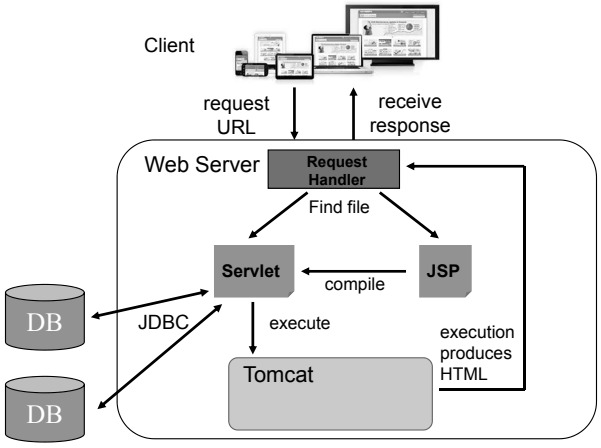
By building the content dynamically, it can have different content for each user. Typically, this dynamic content is retrieved from databases and then inserted into the HTML.

The two standard ways for generating dynamic content using Java are ***Java Server Pages*** (***JSPs***) and ***Servlets***. Since both use Java, they can use JDBC to retrieve dynamic, database content.





## Dynamic Web Server Architecture








## JSP/Servlet Question

***Question:*** **True or False:** Java JSP/Servlet code runs in the browser.

**A)** true

**B)** false

## JSP/Servlet Question #2

***Question:*** **True or False:** JavaScript running in the browser can connect to a database directly using JDBC.

**A)** true

**B)** false





## Java Server Pages (JSPs)

***Java Server Pages*** (***JSPs***) provide a layer above `Servlets` that allow you to mix HTML and calls to Java code.

JSPs are converted into `Servlets`, and are easier to work with for those familiar with HTML.





## Java Server Pages (JSPs) (2)

How to create JSPs:

- 1) Create a file with a `.jsp` extension that contains HTML.
- 2) Include JSP Scripting Elements in the file such as:
  - Expressions `<%= expression %>`
    - Note that an expression returns a string into the HTML.
  - Scriptlets `<% code %>`
  - Declarations `<%! Code %>`
  - Directives `<%@ variable directive %>`
  - Comments `<%-- JSP Comment -->`
- 3) Can use pre-defined variables:
  - `request` `HttpServletRequest`
  - `response` `HttpServletResponse`
  - `session` `HttpSession`
  - `out` `PrintWriter`
    - Prints to HTML document.





## JSP "Hello World!"

```
<html>
<head>
<title>Hello World in JSP</title>
</head>
<body>

<% out.println("Hello World!"); %>

</body>
</html>
```

← Java code embedded in HTML file.





## Useful JSP Code

Include another file:

```
<%@ include file="commontop.html" %>
```

Import from the Java library:

```
<%@ page import="java.util.*" %>
```

Declare and use a variable:

```
<%! private int accessCount = 0 %>
<h2>Accesses to page since server reboot:
<%= ++accessCount %></h2>
```

Access session and request information:

```
<h2>Current time: <%= new java.util.Date() %></h2>
<h2>Remote Host: <%= request.getRemoteHost()%></h2>
<h2>Session ID: <%= session.getID() %></h2>
```





## JSP and JDBC

```
<%@ page import="java.sql.*" %>

<html>
<head><title>Query Results Using JSP</title></head><body>

<%
   Connection con = null;
   try
   {  // Note: Need to define url, uid, pw
      con = DriverManager.getConnection(url, uid, pw);
      Statement stmt = con.createStatement();
      ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                          FROM Emp");
```

## JSP and JDBC (2)

```
        out.print("<table><tr><th>Name</th>");
        out.println("<th>Salary</th></tr>");
        while (rst.next())
        {   out.println("<tr><td>"+rst.getString(1)+"</td>"
                  +"<td>"+rst.getDouble(2)+"</td></tr>");
        }
        out.println("</table>");
        con.close();
    }
    catch (SQLException ex)       { out.println(ex); }
    finally
    {   if (con != null)
            try
            {   con.close(); }
            catch (SQLException ex) { out.println(ex); }
    }
%>
</body>
</html>
```





## Answering Queries using HTML Forms

One of the common uses of dynamic web pages is to construct answers to user queries expressed using HTML forms.

The HTML code will contain a `FORM` and the `FORM ACTION` will indicate the server code to call to process the request.

In our example, the JSP or Servlet gets the HTML request (and parameters in the URL), queries the database, and returns the answers in a table.





## HTTP *GET* and *POST* Methods

`GET` and `POST` are two ways a HTTP client can communicate with a web server. `GET` is used for getting data from the server, and `POST` is used for sending data there.

- `GET` appends the form data (called a query string) to an URL, in the form of key/value pairs, for example, `name=John`.
  - ⇨ In the query string, key/value pairs are separated by & characters, spaces are converted to + characters, and special characters are converted to their hexadecimal equivalents.
  - ⇨ Since the query string is in the URL, the page can be bookmarked. The query string is usually limited to a relatively small amount of data.
- `POST` passes data of unlimited length as an HTTP request body to the server. The user working in the client Web browser cannot see the data that is being sent, so `POST` requests are ideal for sending confidential or large amounts of data to the server.





## *GET* vs. *POST* Question

***Question:*** Select a **true** statement.

**A)** Data sent using `POST` is passed as part of the URL.

**B)** A `GET` request with all parameters cannot be bookmarked.

**C)** Your servlet can do the same action for both `GET` and `POST` requests.

**D)** A `GET` request is used to send a large amount of data.





## HTML Form

```
<html>
<head>
<title>Querying Using JSP/Servlets and Forms</title>
</head>
<body>

<h1>Enter the name and/or department to search for:</h1>

<form method="get"
   action="http://cosc304.ok.ubc.ca/tomcat/EmpQuery.jsp">
Name:<input type="text" name="empname" size="25">
Dept:<input type="text" name="deptnum" size="5">
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</form>

</body>
</html>
```

FORM action and type

FORM buttons and input fields





## Reading Parameters

Parameters passed into the JSP are accessible using the `request` object:

```
String empName = request.getParameter("empname");
String deptNum = request.getParameter("deptnum");
```

## Reading Parameters Badly Building a Query – SQL Injection!

```
String empName = request.getParameter("empname");
String deptNum = request.getParameter("deptnum");
try
{
    con = DriverManager.getConnection(url, uid, pw);
    Statement stmt = con.createStatement();
    String SQLSt = "SELECT ename, salary, dno FROM Emp "
                 + "WHERE 1=1 ";

    if (empName != null && !empName.equals(""))
        SQLSt = SQLSt + " and ename LIKE '%"+empName+"%'";

    if (deptNum != null && !deptNum.equals(""))
        SQLSt = SQLSt + " and dno ='"+deptNum+"'";

    ResultSet rst = stmt.executeQuery(SQLSt);
    ...
```





## Building a Query using a PreparedStatement

```
String empName = request.getParameter("empname");
String deptNum = request.getParameter("deptnum");
try
{
    con = DriverManager.getConnection(url, uid, pw);
    String sql = "SELECT ename, salary, dno FROM Emp";
    boolean hasEmp = empName != null && !empName.equals("");
    boolean hasDept= deptNum != null && !deptNum.equals("");

    PreparedStatement pstmt=null;
    ResultSet rst = null;

    if (!hasEmp && !hasDept)
    {
        pstmt = con.prepareStatement(sql);
        rst = pstmt.executeQuery();
    }
```





## Building a Query using a PreparedStatement

```
    else if (hasEmp)
    {   empName = "%"+empName+"%";
        sql += " WHERE ename LIKE ?";
        if (hasDept)
            sql += " AND dno = ?";
        pstmt = con.prepareStatement(sql);
        pstmt.setString(1, empName);
        if (hasDept)
            pstmt.setString(2, deptNum);
        rst = pstmt.executeQuery();
    }
    else if (hasDept)
    {
        sql += " WHERE dno = ?";
        pstmt = con.prepareStatement(sql);
        pstmt.setString(1, deptNum);
        rst = pstmt.executeQuery();
    }
    ...
```





## JSP Examples Implementing Login and Security

How do you password protect files in your web site?

The basic idea is to:

- Create a login page like `login.jsp`.
- Create a page to validate the user login (often by connecting to the database to determine if given valid user id/password).
- Create a file containing JSP code that is included in every one of your protected pages. This code:
  - ⇨Examines if a session variable flag is set indicating if logged in.
  - ⇨If user is logged in, show page. Otherwise redirect to login page.





## Implementing Login and Security `login.jsp`

```
<html>
<head><title>Login Screen</title></head>
<body>
<center>
<h3>Please Login to System</h3>

<%
// Print prior error login message if present
if  (session.getAttribute("loginMessage") != null)
    out.println("<p>"+
        session.getAttribute("loginMessage").toString()+"</p>");
%>

<br>
```





## Implementing Login and Security `login.jsp (2)`

```
<form name="MyForm" method=post action="validateLogin.jsp">
<table width="40%" border="0" cellspacing="0" cellpadding="0">
<tr>
<td><div align="right">Username:</div></td>
<td><input type="text" name="username" size=8 maxlength=8></td>
</tr>
<tr>
<td><div align="right">Password:</div></td>
<td><input type="password" name="password" size=8
                        maxlength=8"></td>
</tr>
</table>
<input class="submit" type="submit" name="Sub" value="Log In">
</form>
<br/>
</center>
</body>
</html>
```

## Implementing Login and Security
validateLogin.jsp

```
<%@ page language="java" import="java.io.*" %>
<%
    String authenticatedUser = null;
    session = request.getSession(true);// May create new session
    try
    {
        authenticatedUser = validateLogin(out,request,session);
    }
    catch(IOException e)
    {       System.err.println(e); }

    if(authenticatedUser != null)
        response.sendRedirect("protectedPage.jsp"); // Success
    else
        response.sendRedirect("login.jsp"); // Failed login
            // Redirect back to login page with a message
%>
```





## Implementing Login and Security
validateLogin.jsp (2)

```
<%!
String validateLogin(JspWriter out,HttpServletRequest request,
                        HttpSession session) throws IOException
{
    String username = request.getParameter("username");
    String password = request.getParameter("password");
    String retStr = null;

    if(username == null || password == null)
        return null;
    if((username.length() == 0) || (password.length() == 0))
        return null;

    // Should make a database connection here and check password
    // Here just hard-coding password
    if (username.equals("test") && password.equals("test"))
        retStr = username;
```





## Implementing Login and Security
validateLogin.jsp (3)

```
    try {  // Login using database version
           // Make database connection -- Sample query:
        String query = "SELECT pswd FROM User WHERE userID =
         BINARY '"+username+"' AND pswd = BINARY '"+password+"'";
        ResultSet rst = executeQuery(con,query);
        if (!rst.next())    // Such a record does not exist
            retStr = "";
    }
    catch(SQLException e)    { System.err.println(ex); }
    finally { // Close database connection }

    if(retStr != null)
    {   session.removeAttribute("loginMessage");
        session.setAttribute("authenticatedUser",username);
    }
    else
        session.setAttribute("loginMessage","Failed login.");
   return retStr;
} %>
```





## Implementing Login and Security
protectedPage.jsp

```
<html>
<head><title>Password Protected Page</title></head>
<body>

<%@ include file="auth.jsp"%>

<%
String user = (String)session.getAttribute("authenticatedUser");
out.println("<h1>You have access to this page: "+user+"</h1>");
%>

</body>
</html>
```





## Implementing Login and Security
auth.jsp

```
<%
    Object authUser = session.getAttribute("authenticatedUser");
    boolean authenticated = authUser == null ? false : true;

    if (!authenticated)
    {
        String loginMessage = "You have not been authorized to "+
            " access the URL "+request.getRequestURL().toString();
        session.setAttribute("loginMessage",loginMessage);
        response.sendRedirect("login.jsp");
        return;
    }
%>
```





## JSP Examples
## Passing Objects Between Pages

How do you pass information between pages?

One way is to encode that information in the URL as parameters. These parameters can be extracted from the `request` object.

If you have a lot of information, it is better to use the `session` object. The session object allows you to store any number of objects that are later looked up by name. Since they remain on the server, performance/security is improved.

## Passing Objects Between Pages
sendingPage.jsp

```
<%@ page import="java.util.ArrayList,java.util.Random" %>
<html>
<head><title>Sending Data Page</title></head>
<body>

<%
    // Generate and print array
    ArrayList ar = new ArrayList(20);
    Random generator = new Random();
    out.println("<h2>Created the following array:</h2>");
    for (int i = 0; i < 20; i++)
    {   ar.add(new Integer(generator.nextInt(10)));
        out.println(ar.get(i)+"<BR>");
    }
    // Store arraylist in a session variable
    session.setAttribute("arraydata",ar);
%>
</body>
</html>
```





## Passing Objects Between Pages
receivingPage.jsp

```
<%@ page import="java.util.ArrayList" %>
<html>
<head><title>Receiving Data Page</title> </head>
<body>

<%
    ArrayList ar = (ArrayList)session.getAttribute("arraydata");
    if (ar == null)
        out.println("<h2>No data sent to page.</h2>");
    else
    {   out.println("<h2>Received the following array:</h2>");
        for (int i = 0; i < 20; i++)
            out.println(ar.get(i)+"<BR>");
    }
%>
</body>
</html>
```





## Session Question

***Question:*** **True or False:** Data associated with a session remains on the server.

**A)** true

**B)** false





## Servlets

`Servlets` are Java programs that run inside a web server that can perform server-side processing such as interacting with a database or another application. `Servlets` can generate dynamic html and return this to the client.

How to create a `Servlet`:

- 1) create an HTML file that invokes a `Servlet` (usually through the `FORM ACTION=...`).
- 2) create a Java program that does the following:
  - ⇨`import javax.servlet.*;`
  - ⇨`import javax.servlet.http.*;`
  - ⇨inherit from `HttpServlet`
  - ⇨override the `doGet` and `doPost` methods
  - ⇨write the response HTML file using `java.io.Printwriter`





## Servlets Notes

There is one instance of `Servlet` for each `Servlet` name, and the same instance serves all requests to that name.

Instance members persist across all requests to that name.

Local variables in `doPost` and `doGet` are unique to each request.





## Servlets "Hello World!"

```
import javax.servlet.*;
import javax.servlet.http.*;
```
← Import Servlet API

```
public class HelloServlet extends HttpServlet {
    public void init(ServletConfig cfg) throws ServletException {
        super.init(cfg);  // First time Servlet is invoked
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, java.io.IOException {
        doHello(request, response);    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, java.io.IOException {
        doHello(request, response); }

    private void doHello(HttpServletRequest request,HttpServletResponse response)
            throws ServletException, java.io.IOException {
        response.setContentType("text/html");
        java.io.PrintWriter out = response.getWriter();
        out.println("<html><head><title>Hello World</title></head>");
        out.println("<body><h1>Hello World</h1></body></html>");
        out.close();
    }
}
```

Get & Post

← Write out HTML file for client

## Servlets and JDBC

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class JdbcServlet extends HttpServlet {

    private Connection con;

    public void init(ServletConfig cfg) throws ServletException {
        super.init(cfg);

        String url = "<fill-in>";
        con = null;
        try
        {
            con = DriverManager.getConnection(url);
        }
        catch (SQLException e)
        {   throw new ServletException("SQLException: "+e); }
    }
```





## Servlets and JDBC (2)

```
    public void destroy()
    {   try {
            if (con != null)
                con.close();
        }
        catch (SQLException e)
        {   System.err.println("SQLException: "+e); }
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, java.io.IOException
    {
       doTable(request, response);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, java.io.IOException
    {
       doTable(request, response);
    }
```





## Servlets and JDBC (3)

```
    private void doTable(HttpServletRequest request,HttpServletResponse response)
            throws ServletException, java.io.IOException {
        response.setContentType("text/html");
        java.io.PrintWriter out = response.getWriter();
        out.println("<html><head><title></title></head>");

        if (con == null)
            out.println("<body><H1>Unable to connect to DB</H1></body></html>");
        else
        {   try {
                Statement stmt = con.createStatement();
                ResultSet rst = stmt.executeQuery("SELECT ename,salary FROM Emp");

                out.print("<table><tr><th>Name</th><th>Salary</th></tr>");
                while (rst.next())
                {   out.println("<tr><td>"+rst.getString(1)+"</td>"
                                +"<td>"+rst.getDouble(2)+"</td></tr>");
                }
                out.println("</table></body></html>");
                out.close();
            } catch (SQLException ex) { System.err.println(ex); }
        }
    }
}
```





## JSP and Servlets Discussion

Java Server Pages (JSPs) are HTML files with embedded Java code. They are easier to produce because the HTML file is the main product and the Java code is secondary.

When a JSP file is actually used, it is converted to a `Servlet` and run. Apache Tomcat handles the conversion and execution of the `Servlet`.

The advantage of JSP over `Servlets` is that the HTML page can be edited by users not familiar with the Java language using standard HTML editors and the Java code added separately by programmers.





## JavaScript

JavaScript is a *scripting* language used primarily for web pages.

- JavaScript was developed in 1995 and released in the Netscape web browser (since renamed to Mozilla Firefox).

Despite the name, JavaScript is not related to Java, although its syntax is similar to other languages like C, C++, and Java.

- There are some major differences between JavaScript and Java.

From the database perspective, JavaScript is used to make HTML forms more interactive and to validate input client-side before sending it to the server.





## Hello World Example - JavaScript Code `helloWorld.html`

```
<html>
<head>
<title>HelloWorld using JavaScript</title>
</head>

<body>

<h1>

   <script type="text/javascript">
      document.write("Hello, world!");
   </script>
</h1>

</body>
</html>
```

`<script>` tag indicating code

JavaScript code

document is HTML document
document.write() puts that text into the document at this location

## JavaScript and JSP

Your JSP code can either include JavaScript code:

- 1) Directly in the HTML code
- 2) By outputting it using out.println()
  - ⇨With servlets, you only have option #2.

Remember, the JSP/Servlet code is run on the server, and the JavaScript code is run on the client.

- The JavaScript code cannot access the database directly.
- The JSP/Servlet code cannot interact with the user (unless you use AJAX or some other method to send server requests).





## Hello World written by JSP Code
helloWorld_JS.jsp

```
<html>
<head>
<title>HelloWorld using JavaScript</title>
</head>

<body>

<h1>
   <%
      out.println("<script language=\"javascript\">");
      out.println("document.write(\"Hello, world!\")");
      out.println("</script>");
   %>
</h1>

</body>
</html>
```





## JavaScript for Data Validation

JavaScript is commonly used to validate form input on the client-side without sending it to the server. This reduces the load on the server, and more importantly, improves the browsing experience for the user.





## JavaScript Data Validation Example
validateNum.html

```
<html><head><title>Number Field Validation</title></head>
<body>

<script type="text/javascript">
function validate(num){
   if (parseInt(num))
      alert("You entered number: " + parseInt(num));
   else
      alert("Invalid number entered: " + num);
}
</script>

Enter a number:
<form name="input">
   <input type="text" name="numField"
          onchange="validate(this.value)">
</form>

</body>
</html>
```

Validation code

Validate when field is changed. Can also use:
- `onblur` – triggered when field loses focus
- `onsubmit` – triggered when form is submitted





## AJAX

AJAX allows client-side JavaScript to request and receive data from the server without refreshing the page.

***AJAX*** (Asynchronous JavaScript+XML) was named by Jesse Garret in 2005. However, XML is not required for data communication between client and server (JSON is more common).

AJAX uses the **`XMLHttpRequest Object`** to communicate requests and receive results from the server.

Communication can either be synchronous or asynchronous.





## AJAX Validate Form Fields Example
validateUserEmail.html

```
<html><head><title>Validate User Name and Email</title></head>
<body>
<script type="text/javascript">
function checkUserName(){
    var name = document.getElementById("username").value;
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open("GET","validateUserEmail.jsp?name="+name,false);
    xmlhttp.send(null);    // Synchronous version, no additional payload
    if (xmlhttp.responseText != "VALID")
        alert(xmlhttp.responseText);
}
function checkEmail(){
    var email = document.getElementById("email").value;
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.open("GET","validateUserEmail.jsp?email="+email,false);
    xmlhttp.send(null);    // Synchronous version, no additional payload
    if (xmlhttp.responseText != "VALID")
        alert(xmlhttp.responseText);
}
</script>
<form name="input">
    Name: <input type="text" id="username">
        <a href="javascript: checkUserName()">Check</a> <br>
    Email: <input type="text" id="email">
        <a href="javascript: checkEmail()">Check</a>
</form></body></html>
```

# AJAX Validate Form Fields Example
validateUserEmail.jsp

```
<%
    String name = request.getParameter("name");
    if (name != null)
    {   // This is simple validation code.
        if (name.length() < 4)
            out.println("Name too short.  Must be at least 4
                                                  characters.");
        else
            out.println("VALID");
    }
    else
    {
        String email = request.getParameter("email");
        if (email != null)
        {   if (!email.contains("@"))
                out.println("INVALID");
            else
                out.println("VALID");
        }
        else
            out.println("INVALID INPUT");
    }
%>
```





# AJAX Example - Province/State
provinceState.html


```
<html>
<head><title>Province/State</title></head>

<body bgcolor="white">

<form name="input">
    Country: <select id="country" onchange="changeCountry()">
        <option value="CA">Canada</option>
        <option value="US">United States</option>
    </select><br>
    <p id="stateText">State: </p><select id="state"></select>
</form>

<script type="text/javascript">
```





# Province/State for Canada/US
provinceState.html (2)

```
var xmlhttp = new XMLHttpRequest();

function changeCountry_callBack()
{
    if (xmlhttp.readyState == 4 && xmlhttp.status == 200)
    {   var text = xmlhttp.responseText;
        if (text != "INVALID")
        {   var stateList = document.getElementById("state");

            // Remove all current items in list
            for (var i = stateList.length-1; i >= 0; i--)
                stateList.remove(i);

            // Add elements to list
            var values = text.split(',');
            for (var i=0; i < values.length; i++)
            {   var stateCode = values[i];
                var newOpt = document.createElement('option');
                newOpt.text = stateCode;
                newOpt.value = stateCode;

                stateList.add(newOpt, null);
            }
        }
    }
}
```





# Province/State for Canada/US
provinceState.html (3)

```
function changeCountry(){
    var countryList = document.getElementById("country");
    var stateText = document.getElementById("stateText");

    if (countryList.selectedIndex == 1)
        stateText.innerHTML = "State: ";
    else
        stateText.innerHTML = "Province:";

    xmlhttp.open("GET","provinceState.jsp?country="
                                    +countryList.value,true);
    xmlhttp.onreadystatechange = changeCountry_callBack;
    xmlhttp.send(null);    // Asynchronous version
}

changeCountry();       // Initialize list with default as Canada
</script>
</body>
</html>
```





# Province/State for Canada/US – JSP
provinceState.jsp

```
<%@ page import="java.sql.*" %> <%
String country = request.getParameter("country");
if (country == null)
    out.println("INVALID");
else
{   Connection con = null;
    try {
        con = DriverManager.getConnection(url,uid,pw);
        String sql = "select stateCode from states where countrycode=?";
        PreparedStatement pstmt = con.prepareStatement(sql);
        pstmt.setString(1, country);
        ResultSet rst = pstmt.executeQuery();
        StringBuffer buf = new StringBuffer();
        while (rst.next())
        {  buf.append(rst.getString(1));
           buf.append(',');
        }
        if (buf.length() > 0)
           buf.setLength(buf.length()-1);
       out.println(buf.toString());
    }
    catch (SQLException ex){  out.println("INVALID"); }
    finally
    {   if (con != null)
        {   try {   con.close(); }  catch (SQLException ex) {}
        }
    }
}%>
```





# Connection Pools

A ***connection pool*** is a group of database connections managed by a (web) server.

- All connections in the pool are open. Clients request a connection from the pool and return it when done.
- This results in improved performance as connections are shared across clients and do not pay an open/close penalty every time a connection is used.

Using a connection pool in Tomcat with JNDI:

```
Context root = new InitialContext();     // Get root of JNDI tree
String path = "java:comp/env/jdbc/workson";        // Name key
DataSource ds = (DataSource) root.lookup(path);    // JNDI lookup
con = ds.getConnection();  // Get connection from pool
...
con.close();               // Return connection to pool
```

## Connection Pools Configuration

Modify `conf/context.xml` file to add the resource:

```
<Context>
   <Resource name="jdbc/workson" type="javax.sql.DataSource"
             driverClassName="com.mysql.jdbc.Driver"
             url="jdbc:mysql://cs-suse-4.ok.ubc.ca/workson"
             username="rlawrenc" password="pw"
             maxActive="100" maxIdle="30" maxWait="10000"
             auth="Container"/>
</Context>
```

- `name` - following standard convention to add `"jdbc"` but not necessary. Prefix `"java:comp/env"` added by web server (by convention).
- `type` – class/interface of resource
- `driverClassName` – class name of JDBC driver for database
- `url` – connection string for the driver
- `maxActive`, `maxIdle`, `maxWait` – maximum # of connections, idle connection, and time in milliseconds a client will wait before rollback resp.
- `auth` – indicates if resource managed by web server or web app.





## Connection Pools Question

***Question:*** **True or False:** A connection pool will speed up the execution time of a query (not considering connection time).

**A)** true

**B)** false





## Configuring your Web Application

Each web application (*webapp*) has its own directory that stores HTML/JSP files and has a subdirectory `WEB-INF` that contains:

- `classes` directory – stores all Java class files
- `lib` directory – store all jars used by your Servlets
- `web.xml` – is a ***deployment descriptor*** that provides mapping from URL path to Servlet class name. Example:

```
<web-app>
   <servlet>
      <servlet-name>HelloServlet</servlet-name>
      <servlet-class>HelloServlet</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>HelloServlet</servlet-name>
      <url-pattern>/servlet/HelloServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

Internal name used in mapping (you decide)

Servlet Java class name

Internal name (same as line #1)

URL to get to Servlet (from user's perspective)





## PHP

***PHP*** (www.php.net) is a general-purpose scripting language used extensively in web development.

PHP supports several different ways of connecting to databases including a custom MySQL connector as well as support for ODBC and PHP Data Objects (PDO).

Unlike JDBC, each database has its own database extension which has different features and methods.





## PHP MySQLi Example Procedural Version

```
<?php
// Connecting, selecting database
$mysqli = mysqli_connect("cosc304.ok.ubc.ca", "rlawrenc",
                         "<pw>", "db_rlawrenc")
          or die("Could not connect: " . mysql_error());

// Performing SQL query
$query = "SELECT * FROM Emp";
$result = mysqli_query($mysqli, $query);
if (mysqli_connect_errno($mysqli))
  echo "Failed to connect to MySQL: " . mysqli_connect_error();
```





## PHP MySQLi Example Procedural Version (2)

```
// Printing results in HTML
echo "<table>\n";
while ($line = mysqli_fetch_assoc($result)) {
    echo "\t<tr>\n";
    foreach ($line as $col_value) {
        echo "\t\t<td>$col_value</td>\n";
    }
    echo "\t</tr>\n";
}
echo "</table>\n";

// Free resultset
mysqli_free_result($result);

// Closing connection
mysqli_close($mysqli);

?>
```

# PHP MySQLi Example Object-Oriented Version

```
<?php
$mysqli = new mysqli("cosc304.ok.ubc.ca", "rlawrenc", "<pw>",
                     "db_rlawrenc");
if ($mysqli->connect_errno)
  echo "Failed to connect to MySQL: " . $mysqli->connect_error;

$result = $mysqli->query($query);
echo "<table>\n";
while ($line = $result->fetch_assoc()) {
    echo "\t<tr>\n";
    foreach ($line as $col_value) {
        echo "\t\t<td>$col_value</td>\n";
    }
    echo "\t</tr>\n";
}
echo "</table>\n";
$result->free();
$mysqli->close();
?>
```





# Conclusion

JSP and `Servlets` can use JDBC for constructing dynamic web content and answering user queries via HTML.

JSP and `Servlets` are used with HTML forms to allows user to enter queries or information to the database.

- JSP/ `Servlets` can be run using Apache Tomcat which supports connection pooling.

JavaScript is used for browser-based validation and interactivity. AJAX supports requests to server for data.

Connecting and querying a database with PHP (and other languages) is very similar to using Java/JDBC.





# Objectives

- Write a simple JSP page that uses JDBC.
- Explain the relationship between JSP and `Servlets` and how dynamic web pages are created.
- Be able to create client-side code in JavaScript for data validation.
- Explain the general idea with AJAX.
- Explain what a connection pool is and why it is beneficial.
- Be able to write database access code using PHP.

# COSC 304
# Introduction to Database Systems

# Normalization

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Normalization

***Normalization*** is a technique for producing relations with desirable properties.

Normalization decomposes relations into smaller relations that contain less redundancy. This decomposition requires that no information is lost and reconstruction of the original relations from the smaller relations must be possible.

Normalization is a bottom-up design technique for producing relations. It pre-dates ER modeling and was developed by Codd in 1972 and extended by others over the years.

- Normalization can be used after ER modeling or independently.
- Normalization may be especially useful for databases that have already been designed without using formal techniques.





## Normalization Motivation

The purpose of normalization is to develop good relational schemas that minimize redundancies and update anomalies.

***Redundancy*** occurs when the same data value is stored more than once in a relation.

- Redundancy wastes space and reduces performance.

***Update anomalies*** are problems that arise when trying to insert, delete, or update tuples and are often caused by redundancy.

The goal of normalization is to produce a set of relational schemas $R_1$, $R_2$, …, $R_m$ from a set of attributes $A_1$, $A_2$, … ,$A_n$.

- Imagine that the attributes are originally all in one big relation R= {$A_1$, $A_2$, .., $A_n$} which we will call the ***Universal Relation***.
- Normalization divides this relation into $R_1$, $R_2$, …, $R_m$.





## The Power of Normalization Example Relations

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

WorksOn Relation

| eno | pno | resp | hours |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |

Proj Relation

| pno | pname | budget |
|---|---|---|
| P1 | Instruments | 150000 |
| P2 | DB Develop | 135000 |
| P3 | Budget | 250000 |
| P4 | Maintenance | 310000 |
| P5 | CAD/CAM | 500000 |

Dept Relation

| dno | dname | mgreno |
|---|---|---|
| D1 | Management | E8 |
| D2 | Consulting | E7 |
| D3 | Accounting | E5 |
| D4 | Development | null |





## The Power of Normalization Universal Relation

A Universal Relation with all attributes:

| eno | pno | resp | hours | ename | bdate | title | salary | supereno | dno | dname | mgreno | pname | budget |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E1 | P1 | Manager | 12 | J. Doe | 01-05-75 | EE | 30000 | E2 | | | | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 | Instruments | 150000 |
| E2 | P2 | Analyst | 6 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 | DB Develop | 135000 |
| E3 | P3 | Consultant | 10 | A. Lee | 07-05-66 | ME | 40000 | E6 | D2 | Consulting | E7 | Budget | 250000 |
| E3 | P4 | Engineer | 48 | A. Lee | 07-05-66 | ME | 40000 | E6 | D2 | Consulting | E7 | Maintenance | 310000 |
| E4 | P2 | Programmer | 18 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 | DB Develop | 135000 |
| E5 | P2 | Manager | 24 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | J. Jones | 10-11-72 | SA | 50000 | | D1 | Management | E8 | Budget | 250000 |

Universal(<u>eno</u>, <u>pno</u>, resp, hours, ename, bdate, title, salary, supereno, dno, dname, mgreno, pname, budget)

What are some of the problems with the Universal Relation?

Normalization will allow us to get back to the starting relations.





## Update Anomalies

There are three major types of update anomalies:

- ***Insertion Anomalies*** - Insertion of a tuple into the relation either requires insertion of redundant information or cannot be performed without setting key values to `NULL`.
- ***Deletion Anomalies*** - Deletion of a tuple may lose information that is still required to be stored.
- ***Modification Anomalies*** - Changing an attribute of a tuple may require changing multiple attribute values in other tuples.

# Example Relation Instances

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

Dept Relation

| dno | dname | mgreno |
|---|---|---|
| D1 | Management | E8 |
| D2 | Consulting | E7 |
| D3 | Accounting | E5 |
| D4 | Development | null |

EmpDept Relation

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null | null | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 | Management | E8 |





# Insertion Anomaly Example

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null | null | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 | Management | E8 |

Consider these two types of insertion anomalies:

- 1) Insert a new employee E9 working in department D2.
  - ⇨ You have to redundantly insert the department name and manager when adding this record.
- 2) Insert a department D4 that has no current employees.
  - ⇨ This insertion is not possible without creating a dummy employee id and record because `eno` is the primary key of the relation.





# Deletion Anomaly Example

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null | null | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 | Management | E8 |

Consider this deletion anomaly:

- Delete employees E3 and E6 from the database.
- Deleting those two employees removes them from the database, and we now have lost information about department D2!





# Modification Anomaly Example

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null | null | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 | Management | E8 |

Consider these modification anomalies:

- 1) Change the name of department D3 to Embezzling.
  - ⇨ You must update the department name in 3 different records.
- 2) Change the manager of D1 to E4.
  - ⇨ You must update the `mgreno` field in 2 different records.





# Desirable Relational Schema Properties

Relational schemas that are well-designed have several important properties:

- 1) The most basic property is that relations consists of attributes that are logically related.
  - ⇨ The attributes in a relation should belong to only one entity or relationship.
- 2) ***Lossless-join property*** ensures that the information decomposed across many relations can be reconstructed using natural joins.
- 3) ***Dependency preservation property*** ensures that constraints on the original relation can be maintained by enforcing constraints on the normalized relations.





# Normalization Question

***Question:*** How many of the following statements are ***true***?

**1)** Normalization is a bottom up process.

**2)** Anomalies with updates often are indicators of a bad design.

**3)** The lossless-join property means that it is possible to reconstruct the original relation after normalization using joins.

**4)** The dependency preservation property means that constraints should be preserved before and after normalization.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4

# Functional Dependencies

Functional dependencies represent constraints on the values of attributes in a relation and are used in normalization.

A ***functional dependency*** (abbreviated ***FD***) is a statement about the relationship between attributes in a relation. We say a set of attributes *X* functionally determines an attribute *Y* if given the values of *X* we always know the only possible value of *Y*.

- Notation: $X \rightarrow Y$
- *X* functionally determines *Y*
- *Y* is functionally dependent on *X*

Example:

- eno $\rightarrow$ ename
- eno, pno $\rightarrow$ hours





# Notation for Functional Dependencies

A functional dependency has a left-side called the ***determinant*** which is a set of attributes, and one attribute on the right-side.

$$\text{eno, pno} \rightarrow \text{hours}$$

determinant — determined attribute

Strictly speaking, there is always only one attribute on the RHS, but we can combine several functional dependencies into one:

$$\text{eno, pno} \rightarrow \text{hours}$$
$$\text{eno, pno} \rightarrow \text{resp}$$
$$\text{eno, pno} \rightarrow \text{hours, resp}$$

Remember that this is really short-hand for two functional dependencies.





# The Semantics of Functional Dependencies

Functional dependencies are a property of the ***domain*** being modeled **NOT** of the data instances currently in the database.

- This means that similar to keys you cannot tell if one attribute is functionally dependent on another by looking at the data.

Example:

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

- List the functional dependencies of the attributes in this relation.





# The Semantics of Functional Dependencies (2)

Functional dependencies are directional.

eno $\rightarrow$ ename does not mean that ename $\rightarrow$ eno

Example:

Emp Relation

| eno | ename | bdate | title | salary | supereno | dno |
|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 |

- Given an employee name there may be multiple values for `eno` if we have employees in the database with the same name.
- Thus knowing `ename`, does not uniquely tell us the value of `eno`.





# Trivial Functional Dependencies

A functional dependency is ***trivial*** if the attributes on its left-hand side are a superset of the attributes on its right-hand side.

Examples: eno $\rightarrow$ eno

eno, ename $\rightarrow$ eno

eno, pno, hours $\rightarrow$ eno, hours

Trivial functional dependencies are not interesting because they do not tell us anything.

- Trivial FDs basically say "If you know the values of these attributes, then you uniquely know the values of any subset of those attributes.

We are only interested in ***nontrivial*** FDs.





# Identifying Functional Dependencies

Identify all non-trivial functional dependencies in the Universal Relation:

| eno | pno | resp | hours | ename | bdate | title | salary | supereno | dno | dname | mgreno | pname | budget |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E1 | P1 | Manager | 12 | J. Doe | 01-05-75 | EE | 30000 | E2 | | | | Instruments | 150000 |
| E2 | P1 | Analyst | 24 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 | Instruments | 150000 |
| E2 | P2 | Analyst | 6 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 | DB Develop | 135000 |
| E3 | P3 | Consultant | 10 | A. Lee | 07-05-66 | ME | 40000 | E6 | D2 | Consulting | E7 | Budget | 250000 |
| E3 | P4 | Engineer | 48 | A. Lee | 07-05-66 | ME | 40000 | E6 | D2 | Consulting | E7 | Maintenance | 310000 |
| E4 | P2 | Programmer | 18 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 | DB Develop | 135000 |
| E5 | P2 | Manager | 24 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 | DB Develop | 135000 |
| E6 | P4 | Manager | 48 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 | Maintenance | 310000 |
| E7 | P3 | Engineer | 36 | J. Jones | 10-11-72 | SA | 50000 | | D1 | Management | E8 | Budget | 250000 |

## Why the Name "Functional" Dependencies?

Functional dependencies get their name because you could imagine the existence of some function that takes in the parameters of the left-hand side and computes the value on the right-hand side of the dependency.

Example: eno, pno → hours

f(eno, pno) → hours

```
int f (String eno, String pno)
{       // Do some lookup...
        return hours;
}
```

Remember that no such function exists, but it may be useful to think of FDs this way.





## Functional Dependencies and Keys

Functional dependencies can be used to determine the candidate and primary keys of a relation.

- For example, if an attribute functionally determines all other attributes in the relation, that attribute can be a key:

eno → eno, ename, bdate, title, supereno, dno

- `eno` is a candidate key for the `Employee` relation.

Alternate definition of keys:

- A set of attributes *K* is a ***superkey*** for a relation *R* if the set of attributes *K* functionally determines all attributes in *R*.
- A set of attributes *K* is a ***candidate key*** for a relation *R* if *K* is a *minimal* superkey of *R*.





## Functional Dependencies and Keys (2)

EmpDept Relation

| eno | ename | bdate | title | salary | supereno | dno | dname | mgreno |
|---|---|---|---|---|---|---|---|---|
| E1 | J. Doe | 01-05-75 | EE | 30000 | E2 | null | null | null |
| E2 | M. Smith | 06-04-66 | SA | 50000 | E5 | D3 | Accounting | E5 |
| E3 | A. Lee | 07-05-66 | ME | 40000 | E7 | D2 | Consulting | E7 |
| E4 | J. Miller | 09-01-50 | PR | 20000 | E6 | D3 | Accounting | E5 |
| E5 | B. Casey | 12-25-71 | SA | 50000 | E8 | D3 | Accounting | E5 |
| E6 | L. Chu | 11-30-65 | EE | 30000 | E7 | D2 | Consulting | E7 |
| E7 | R. Davis | 09-08-77 | ME | 40000 | E8 | D1 | Management | E8 |
| E8 | J. Jones | 10-11-72 | SA | 50000 | null | D1 | Management | E8 |

eno → eno, ename, bdate, title, salary, supereno, dno

dno → dname, mgreno

Question: List the candidate key(s) of this relation.





## Functional Dependency Rules

Functional dependencies follow rules (Armstrong's Axioms 1974) with the most important being ***transitivity***.

**Transitive Rule**: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$





## Functional Dependency Question

***Question:*** How many of the following statements are ***true***?

**1)** Functional dependencies are not directional.

**2)** A trivial functional dependency has its right side as a subset of (or the same as) its left side.

**3)** A key is a set of attributes that functionally determines all attributes in a relation.

**4)** Functional dependencies can be determined by examining the data in a relation.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Computing the Closure of FDs

The transitivity rule of FDs can be used for three purposes:

- 1) To determine if a given FD $X \rightarrow Y$ follows from a set of FDs F.
- 2) To determine if a set of attributes *X* is a superkey of *R*.
- 3) To determine the set of all FDs (called the ***closure*** $F^+$) that can be inferred from a set of initial functional dependencies *F*.

The basic idea is that given any set of attributes *X,* we can compute the set of all attributes $X^+$ that can be functionally determined using *F*. This is called the ***closure of X under F***.

- For purpose #1, we know that $X \rightarrow Y$ holds if Y is in $X^+$.
- For purpose #2, *X* is a superkey of *R* if $X^+$ is all attributes of R.
- For purpose #3, we can compute $X^+$ for all possible subsets X of R to derive all FDs (the ***closure*** $F^+$) .

## Computing the Attribute Closure

The algorithm is as follows:

- Given a set of attributes $X$.
- Let $X^+ = X$
- Repeat
  - ⇨Find a FD in $F$ whose left side is a subset of $X^+$.
  - ⇨Add the right side of $F$ to $X^+$.
- Until ($X^+$ does not change)

After the algorithm completes you have a set of attributes $X^+$ that can be functionally determined from $X$. This allows you to produce FDs of the form:

- $X \rightarrow A$ where $A$ is in $X^+$





## Computing Attribute Closure Example

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

$F = \{A \rightarrow B,C \ , \ C \rightarrow D \ , \ D \rightarrow G\}$

Compute $\{A\}^+$:

- $\{A\}^+ = \{A\}$ (initial step)
- $\{A\}^+ = \{A,B,C\}$ (by using FD $A \rightarrow B,C$)
- $\{A\}^+ = \{A,B,C,D\}$ (by using FD $C \rightarrow D$)
- $\{A\}^+ = \{A,B,C,D,G\}$ (by using FD $D \rightarrow G$)

Similarly we can compute $\{C\}^+$ and $\{E,G\}^+$:

- $\{C\}^+ = \{C,D,G\}$
- $\{E,G\}^+ = \{E,G\}$





## Using Attribute Closure to Detect a FD

The attribute closure algorithm can be used to determine if a particular FD $X \rightarrow Y$ holds based on the set of given FDs $F$.

- Compute $X^+$ and see if Y is in $X^+$.

Example:

- $R = (A,B,C,D,E,G)$
- $F = \{A \rightarrow B,C \ , \ C \rightarrow D \ , \ D \rightarrow G\}$
- Does the FD $C,D \rightarrow G$ hold?
  - ⇨Compute $\{C,D\}^+ = \{C,D,G\}$. Yes, the FD $C,D \rightarrow G$ holds.
- Does the FD $B,C \rightarrow E$ hold?
  - ⇨Compute $\{B,C\}^+ = \{B,C,D,G\}$. No, the FD $B,C \rightarrow E$ does not hold.





## Function Dependency Closure Question

***Question:*** Given the following, does $\{A,B \rightarrow D\}$ hold?

$R = (A,B,C,D)$

$F = \{ A,B \rightarrow C \ , \ C \rightarrow D \ , \ D \rightarrow A \ \}$

**A)** yes
**B)** no





## FD Closure Question 2

***Question:*** Given the following, does $\{B,D \rightarrow A,C\}$ hold?

$R = (A,B,C,D)$

$F = \{A,B \rightarrow C \ , \ B,C \rightarrow D \ , \ C,D \rightarrow A \ , \ A,D \rightarrow B\}$

**A)** yes
**B)** no





## Function Dependency Key Question

***Question:*** Given the following FDs, how many keys of R?

$R = (A,B,C,D)$

$F = \{ A,B \rightarrow C \ , \ C \rightarrow D \ , \ D \rightarrow A \ \}$

**A)** 0
**B)** 1
**C)** 2
**D)** 3
**E)** 4

## Function Dependency Key Question 2

***Question:*** Given the following FDs, how many keys of R?

R = (A,B,C,D)

F = {A,B → C , B,C → D , C,D → A ,
A,D → B}

**A)** 0
**B)** 1
**C)** 2
**D)** 3
**E)** 4





## Minimal Set of FDs

A set of FDs *F* is ***minimal*** if it satisfies these conditions:

- Every FD in *F* has a single attribute on its right-hand side.
- We cannot replace any FD $X \rightarrow A$ in *F* with $Y \rightarrow A$ where $Y \subset X$ and still have a set of dependencies that is equivalent to *F*.
- We cannot remove any FD and still have a set that is equivalent to *F*.

A minimal set of FDs is like a canonical form of FDs.

Note that there may be many *minimal covers* for a given set of functional dependencies *F*.





## Full Functional Dependencies

A functional dependency $A \rightarrow B$ is a ***full functional dependency*** if removal of any attribute from *A* results in the dependency not existing any more.

- We say that *B* is *fully functionally dependent* on *A*.
- If remove an attribute from *A* and the functional dependency still exists, we say that *B* is partially dependent on *A*.

Example:

| | |
|---|---|
| eno → ename | (full FD) |
| eno, ename → salary, title | (partial FD - can remove ename) |
| eno, pno → hours, resp | (full FD) |





## Full Functional Dependency Question

***Question:*** **True or False:** You can determine if certain, valid functional dependencies are full functional dependencies without having any domain knowledge.

**A)** true

**B)** false





## Normal Forms

A relation is in a particular ***normal form*** if it satisfies certain normalization properties.

There are several normal forms defined:

- 1NF - First Normal Form
- 2NF - Second Normal Form
- 3NF - Third Normal Form
- BCNF - Boyce-Codd Normal Form
- 4NF - Fourth Normal Form
- 5NF - Fifth Normal Form

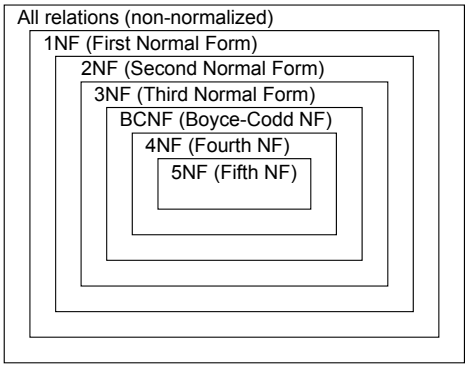
Each of these normal forms are stricter than the next.

⇨ For example, 3NF is better than 2NF because it removes more redundancy/anomalies from the schema than 2NF.





## Normal Forms



# First Normal Form (1NF)

A relation is in ***first normal form*** (***1NF***) if all its attribute values are atomic.

That is, a 1NF relation cannot have an attribute value that is:

- a set of values (multi-valued attribute)
- a set of tuples (nested relation)

1NF is a standard assumption in relational DBMSs.

- However, object-oriented DBMSs and nested relational DBMSs relax this constraint.

A relation that is not in 1NF is an ***unnormalized*** relation.





# A non-1NF Relation

| eno | ename | pno | resp | hours |
|---|---|---|---|---|
| E1 | J. Doe | P1 | Manager | 12 |
| E2 | M. Smith | P1 | Analyst | 24 |
| | | P2 | Analyst | 6 |
| E3 | A. Lee | P3 | Consultant | 10 |
| | | P4 | Engineer | 48 |
| E4 | J. Miller | P2 | Programmer | 18 |
| E5 | B. Casey | P2 | Manager | 24 |
| E6 | L. Chu | P4 | Manager | 48 |
| E7 | J. Jones | P3 | Engineer | 36 |

Two equivalent representations

| eno | ename | pno | resp | hours |
|---|---|---|---|---|
| E1 | J. Doe | P1 | Manager | 12 |
| E2 | M. Smith | {P1,P2} | {Analyst,Analyst} | {24,6} |
| E3 | A. Lee | {P3,P4} | {Consultant,Engineer} | {10,48} |
| E4 | J. Miller | P2 | Programmer | 18 |
| E5 | B. Casey | P2 | Manager | 24 |
| E6 | L. Chu | P4 | Manager | 48 |
| E7 | J. Jones | P3 | Engineer | 36 |

Two ways to convert a non-1NF relation to a 1NF relation:

- 1) ***Splitting Method*** - Divide the existing relation into two relations: non-repeating attributes and repeating attributes.
  - ⇨Make a relation consisting of the primary key of the original relation and the repeating attributes. Determine a primary key for this new relation.
  - ⇨Remove the repeating attributes from the original relation.
- 2) ***Flattening Method*** - Create new tuples for the repeating data combined with the data that does not repeat.
  - ⇨Introduces redundancy that will be later removed by normalization.
  - ⇨Determine primary key for this flattened relation.





# Converting a non-1NF Relation to 1NF Using Splitting

| eno | ename | pno | resp | hours |
|---|---|---|---|---|
| E1 | J. Doe | P1 | Manager | 12 |
| E2 | M. Smith | P1 | Analyst | 24 |
| | | P2 | Analyst | 6 |
| E3 | A. Lee | P3 | Consultant | 10 |
| | | P4 | Engineer | 48 |
| E4 | J. Miller | P2 | Programmer | 18 |
| E5 | B. Casey | P2 | Manager | 24 |
| E6 | L. Chu | P4 | Manager | 48 |
| E7 | J. Jones | P3 | Engineer | 36 |

Also need original primary key: eno

Repeating group: (pno, resp, hours)

| eno | ename |
|---|---|
| E1 | J. Doe |
| E2 | M. Smith |
| E3 | A. Lee |
| E4 | J. Miller |
| E5 | B. Casey |
| E6 | L. Chu |
| E7 | J. Jones |

| eno | pno | resp | hours |
|---|---|---|---|
| E1 | P1 | Manager | 12 |
| E2 | P1 | Analyst | 24 |
| E2 | P2 | Analyst | 6 |
| E3 | P3 | Consultant | 10 |
| E3 | P4 | Engineer | 48 |
| E4 | P2 | Programmer | 18 |
| E5 | P2 | Manager | 24 |
| E6 | P4 | Manager | 48 |
| E7 | P3 | Engineer | 36 |





# Converting a non-1NF Relation to 1NF Using Flattening

| eno | ename | pno | resp | hours |
|---|---|---|---|---|
| E1 | J. Doe | P1 | Manager | 12 |
| E2 | M. Smith | P1 | Analyst | 24 |
| | | P2 | Analyst | 6 |
| E3 | A. Lee | P3 | Consultant | 10 |
| | | P4 | Engineer | 48 |
| E4 | J. Miller | P2 | Programmer | 18 |
| E5 | B. Casey | P2 | Manager | 24 |
| E6 | L. Chu | P4 | Manager | 48 |
| E7 | J. Jones | P3 | Engineer | 36 |

| eno | ename | pno | resp | hours |
|---|---|---|---|---|
| E1 | J. Doe | P1 | Manager | 12 |
| E2 | M. Smith | P1 | Analyst | 24 |
| E2 | M. Smith | P2 | Analyst | 6 |
| E3 | A. Lee | P3 | Consultant | 10 |
| E3 | A. Lee | P4 | Engineer | 48 |
| E4 | J. Miller | P2 | Programmer | 18 |
| E5 | B. Casey | P2 | Manager | 24 |
| E6 | L. Chu | P4 | Manager | 48 |
| E7 | J. Jones | P3 | Engineer | 36 |





# Second Normal Form (2NF)

A relation is in ***second normal form*** (***2NF***) if it is in 1NF and every non-prime attribute is fully functionally dependent on a candidate key.

- A ***prime attribute*** is an attribute in any candidate key.

If there is a FD X $\rightarrow$ Y that violates 2NF:

- Compute $X^+$.
- Replace $R$ by relations: $R_1 = X^+$ and $R_2 = (R - X^+) \cup X$

Note:

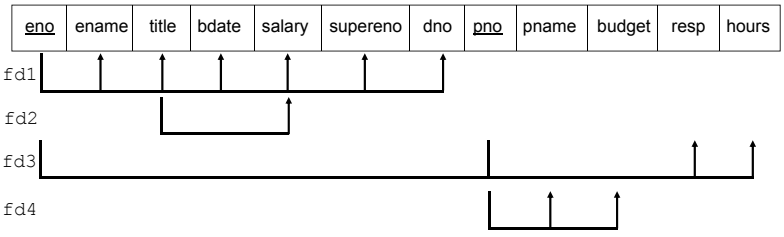
- By definition, any relation with a single key attribute is in 2NF.





# Second Normal Form (2NF) Example

EmpProj relation:

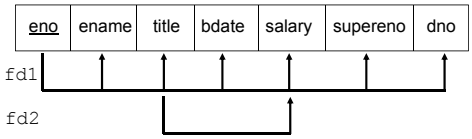



fd1 and fd4 are partial functional dependencies. Normalize to:

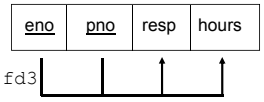
- Emp (eno, ename, title, bdate, salary, supereno, dno)
- WorksOn (eno, pno, resp, hours)
- Proj (pno, pname, budget)

## Second Normal Form (2NF) Example (2)

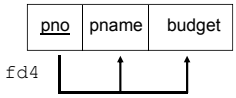
Emp relation:




WorksOn relation:




Proj relation:








## Third Normal Form (3NF)

A relation is in ***third normal form*** (***3NF***) if it is in 2NF and there is no non-prime attribute that is transitively dependent on the primary key.

That is, for all functional dependencies $X \rightarrow Y$ of $R$, one of the following holds:

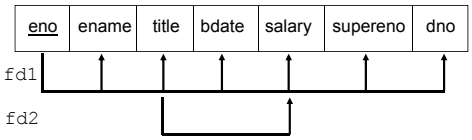
- Y is a prime attribute of R
- X is a superkey of R





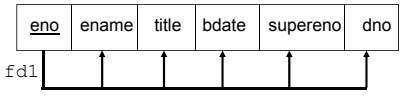
## Third Normal Form (3NF) Example

Emp relation:

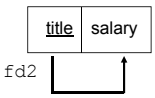



fd2 results in a transitive dependency eno $\rightarrow$ salary. Remove it.

Emp




Pay








## Normalization Question

Consider the universal relation R(A,B,C,D,E,F,G,H,I,J) and the set of functional dependencies:

- F= { $A,B \rightarrow C$ ; $A \rightarrow D,E$ ; $B \rightarrow F$ ; $F \rightarrow G,H$ ; $D \rightarrow I,J$ }

List the keys for $R$.

Decompose $R$ into 2NF and then 3NF relations.





## Boyce-Codd Normal Form (BCNF)

A relation is in ***Boyce-Codd normal form*** (***BCNF***) if and only if every determinant of a non-trivial FD is a superkey.

To test if a relation is in BCNF, we take the determinant of each non-trivial FD in the relation and determine if it is a superkey.

The difference between 3NF and BCNF is that 3NF allows a FD $X \rightarrow Y$ to remain in the relation if $X$ is a superkey **or** $Y$ is a prime attribute. BCNF only allows this FD if $X$ is a superkey.

- Thus, BCNF is more restrictive than 3NF. However, in practice most relations in 3NF are also in BCNF.

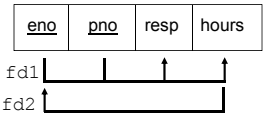




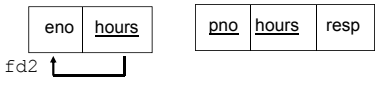
## Boyce-Codd Normal Form (BCNF) Example

Consider the `WorksOn` relation where we have the added constraint that given the hours worked, we know exactly the employee who performed the work. (i.e. each employee is FD from the hours that they work on projects). Then:

WorksOn




Relation is in 3NF but not BCNF.




Note that we lose the FD eno,pno $\rightarrow$ resp, hours.

# BCNF versus 3NF

We can decompose to BCNF but sometimes we do not want to if we lose a FD.

The decision to use 3NF or BCNF depends on the amount of redundancy we are willing to accept and the willingness to lose a functional dependency.

Note that we can always preserve the lossless-join property (recovery) with a BCNF decomposition, but we do not always get dependency preservation.

In contrast, we get both recovery and dependency preservation with a 3NF decomposition.

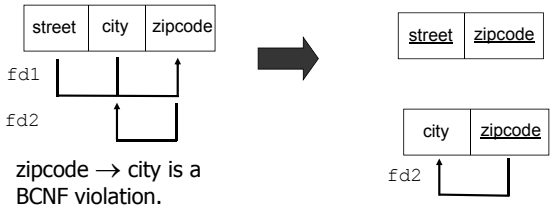




# BCNF versus 3NF Example

An example of not having dependency preservation with BCNF:

- street,city → zipcode and zipcode → city
- Two keys: {street,city} and {street, zipcode}




zipcode → city is a BCNF violation.





# BCNF versus 3NF Example (2)

Consider an example instance:

| | street | zip |
|---|---|---|
| invalid | 545 Tech Sq. | 02138 |
| MIT | 545 Tech Sq. | 02139 |

| city | zip | |
|---|---|---|
| Cambridge | 02138 | Harvard |
| Cambridge | 02139 | MIT |

Join tuples with equal zipcodes:

| street | city | zip | |
|---|---|---|---|
| 545 Tech Sq. | Cambridge | 02138 | invalid |
| 545 Tech Sq. | Cambridge | 02139 | MIT |

Note that the decomposition did not allow us to enforce the constraint that street,city → zipcode even though no FDs were violated in the decomposed relations.





# Conversion to BCNF

Algorithm for converting to BCNF given relation *R* with FDs *F*:

- Look among the given FDs for a BCNF violation $X \rightarrow Y$.
- Compute $X^+$.
  - ⇨ Note that $X^+$ cannot be the set of all attributes or else $X$ is a superkey.
- Replace R by relations with schemas:
  - ⇨ $R_1 = X^+$
  - ⇨ $R_2 = (R - X^+) \cup X$
- Project the FDs *F* onto the two new relations by:
  - ⇨ Computing the closure of *F* and then using only the FDs whose attributes are all in $R_1$ or $R_2$.

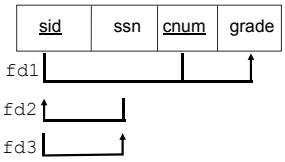




# Normalization to BCNF Question

Given this schema normalize into BCNF directly.

StudentCourse



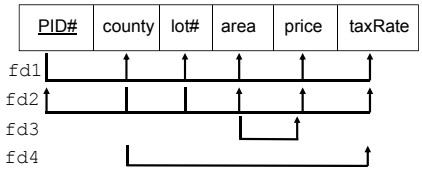





# Normalization Question

Given this database schema normalize into BCNF.

Lots

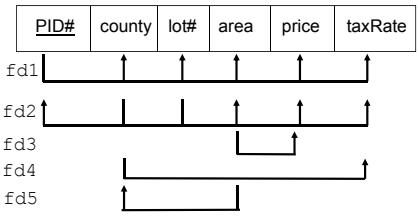



- Description:
  - ⇨ Lots (parcels of land) are identified within each county.
  - ⇨ Each lot can be uniquely identified either by the Property ID# or by the County name and the Lot#.
    - property id's are unique but lot#'s are unique only with the county
  - ⇨ The tax rate is constant within each county.
  - ⇨ The price is based on the area of the land.

## Normalization Question 2

Given this database schema normalize into BCNF.

Lots




- New FD5 says that the size of the parcel of land determines what county it is in.





## Normalization to BCNF Question

Given this schema normalize into BCNF:

- R (courseNum, secNum, offeringDept, creditHours, courseLevel, instructorSSN, semester, year, daysHours, roomNum, numStudents)
- courseNum → offeringDept,creditHours, courseLevel
- courseNum, secNum, semester, year → daysHours, roomNum, numStudents, instructorSSN
- roomNum, daysHours, semester, year → instructorSSN, courseNum, secNum





## Fourth Normal Form (4NF) and Fifth Normal Form (5NF)

Fourth normal form (4NF) and fifth normal formal (5NF) are rarely used in practice.

A relation is in ***fourth normal form*** (***4NF***) if it is in BCNF and contains no non-trivial multi-valued dependencies.

- A multi-valued dependency (MVD) occurs when two independent, multi-valued attributes are present in the schema.

A relation is in ***fifth normal form*** (***5NF***) if the relation has no join dependency.

- A join dependency implies that spurious tuples are generated when the relations are natural joined.





## Normal Forms in Practice

Normal forms are used to prevent anomalies and redundancy. However, just because successive normal forms are better in reducing redundancy that does not mean they always have to be used.

For example, query execution time may increase because of normalization as more joins become necessary to answer queries.

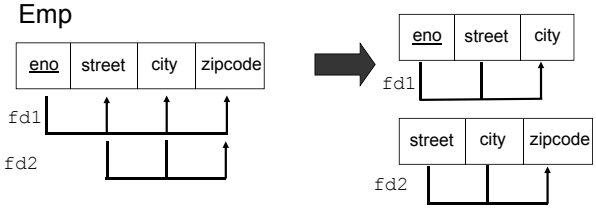




## Normal Forms in Practice Example

For example, `street` and `city` uniquely determine a `zipcode`.

Emp




In this case, reducing redundancy is not as important as the fact that a join is necessary every time the `zipcode` is needed.

- When a `zipcode` does change, it is easy to scan the entire `Emp` relation and update it accordingly.





## Normal Forms and ER Modeling

Normalization and ER modeling are two independent concepts.

You can use ER modeling to produce an initial relational schema and then use normalization to remove any remaining redundancies.

- If you are a good ER modeler, it is rare that much normalization will be required.

In theory, you can use normalization by itself. This would involve identifying all attributes, giving them unique names, discovering all FDs and MVDs, then applying the normalization algorithms.

- Since this is a lot harder than ER modeling, most people do not do it.

## Normal Forms in Practice Summary

In summary, normalization is typically used in two ways:

- To improve on relational schemas after ER design.
- To improve on an existing relational schemas that are poorly designed and contain redundancy and potential for anomalies.

In practice, most designers make sure their schemas are in at least 3NF or BCNF because this removes most anomalies and redundancies. If multi-valued dependencies exist, they should definitely be removed to go to 4NF.

There is always a tradeoff in normalization: Normalization reduces redundancy but fragments the relations which makes it more expensive to query. Some redundancy may help performance.

- This is the classic time versus space tradeoff!





## Conclusion

***Normalization*** is produces relations with desirable properties and reduces redundancy and update anomalies.

Normal forms indicate when relations satisfy certain properties.

- 1NF - All attributes are atomic.
- 2NF - All attributes are fully functionally dependent on a key.
- 3NF - There are no transitive dependencies in the relation.
- 4NF - There are no multi-valued dependencies in the relation.
- 5NF - The relation has no join dependency.

In practice, normalization is used to improve schemas produced after ER design and existing relational schemas.

- Full normalization is not always beneficial as it may increase query time.





## Objectives

- Explain process of normalization and why it is beneficial.
- Describe the concept of the Universal Relation.
- What are the motivations for normalization?
- What are the 3 types of update anomalies? Be able to give examples.
- Describe the lossless-join property and dependency preservation property.
- Define and explain the concept of a functional dependency.
- What is a trivial FD?
- Describe the relationship between FDs and keys.
- Be able to compute the closure of a set of attributes.





## Objectives (2)

- Define normal forms, be able to list the normal forms, and draw a diagram showing their relationship.
- Define 1NF. Know how to convert non-1NF relation to 1NF.
- Define 2NF. Convert 1NF relation to 2NF.
- Define 3NF. Convert 2NF relation to 3NF.
- Define BCNF. Convert 3NF relation to BCNF. Compare BCNF and 3NF. Be able to convert directly to BCNF using algorithm.
- Explain normalization in terms of the time versus space tradeoff.
- Explain the relationship between normalization and ER modeling.

# COSC 304
# Introduction to Database Systems

# JavaScript Object Notation (JSON)

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## JavaScript Object Notation (JSON)

***JavaScript Object Notation (JSON)*** is a method for serializing data objects into text form.

Benefits:

- Human-readable
- Supports semi-structured data
- Supported by many languages (not just JavaScript)

Often used for data interchange especially with AJAX/REST from web server to client.





## JSON Example

JSON constructs:

- **Values:** number, strings (double quoted), `true`, `false`, `null`
- **Objects:** enclosed in `{ }` and consist of set of key-value pairs
- **Arrays:** enclosed in `[ ]` and are lists of values
- Objects and arrays can be nested.

Example:

```
{"Employees": [{"eno":"E1", "ename":"J. Doe", "title":"EE", "salary":30000, "WorksOn":["P1"]},
               {"eno":"E2", "ename":"M. Smith", "title":"SA", "salary":50000, "WorksOn":["P1","P2"]},
               {"eno":"E3", "ename":"A. Lee", "title":"ME", "salary":40000, "WorksOn":["P3"]}
],
"Projects": [{"pno":"P1", "pname":"Instruments", "budget":150000},
             {"pno":"P2", "pname":"DB Develop", "budget":135000},
             {"pno":"P3", "pname":"Budget", "budget":250000}
]
}
```





## JSON versus Relations

| | JSON | Relational |
|---|---|---|
| **Structure** | Nested objects + arrays | Tables |
| **Schema** | Variable (and not required) | Fixed |
| **Queries** | Limited | SQL, RA |
| **Ordering** | Arrays are sorted | No |
| **Systems** | Used with programming languages and some NoSQL systems | Many commercial and open source systems |





## JSON Parsers

A **JSON parser** converts a JSON file (or string) into program objects assuming no syntactic errors.

- In JavaScript, can call `eval()` method on variable containing a JSON string.

A **JSON validator** validates according to a schema and then performs the parsing.

Online validation tool: http://jsonlint.com





## Using JSON in Programs

Many programming languages have APIs to allow for the creation and manipulation of JSON.

One common usage is for the JSON data to be provided from a server (either from a relational or NoSQL database) and sent to a web client.

The web client then uses JavaScript to convert the JSON into objects and manipulate it as required.

## JSON Question

***Question:*** **True or False:** The following JSON is valid.

```
{"employee":1}
```

**A)** true
**B)** false





## JSON Question (2)

***Question:*** **True or False:** The following JSON is valid.

```
{"array":["a", 1, {"c":2}]}
```

**A)** true
**B)** false





## JSON Question (3)

***Question:*** **True or False:** The following JSON is valid.

```
{"array":["a", true, FALSE]}
```

**A)** true
**B)** false





## JSON Question (4)

***Question:*** **True or False:** The following JSON is valid.

```
{}
```

**A)** true
**B)** false





## JSON Question (5)

***Question:*** **True or False:** The following JSON is valid.

```
{4, 5, "c":"a"}
```

**A)** true
**B)** false





## Conclusion

***JavaScript Object Notation (JSON)*** is a method for serializing data objects into text form and is commonly used for data interchange.

JSON uses base values, objects, and arrays to encode data and nesting is possible.

JSON is semi-structured similar to XML but is often regarded as "simpler" and is widely supported by browsers and programming languages.

# *Objectives*

Understand the basic constructs used to encode JSON data.

Compare JSON representation versus relational model.

Practice using JSON in programs.

# COSC 304
# Introduction to Database Systems

# NoSQL Databases

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Relational Databases

Relational databases are the dominant form of database and apply to many data management problems.

- Over $25 billion annual market in 2015.

Relational databases are not the only way to represent data and not the best way for some problems.

Other models:

- Hierarchical model
- Object-oriented
- XML
- Graphs
- Key-value stores
- Document models





## Relational Databases Challenges

Some features of relational databases make them "challenging" for certain problems:

- 1) Fixed schemas – The schemas must be defined ahead of time, changes are difficult, and lots of real-world data is "messy".
  - ⇨ Solution: Get rid of the schemas! Who wants to do that design work anyways? Will you miss them?
- 2) Complicated queries – SQL is declarative and powerful but may be overkill.
  - ⇨ Solution: Simple query mechanisms and do a lot of work in code.
- 3) Transaction overhead – Not all data and query answers need to be perfect. "Close enough is sometimes good enough".
- 4) Scalability – Relational databases may not scale sufficiently to handle high data and query loads or this scalability comes with a very high cost.





## NoSQL

NoSQL databases are useful for several problems not well-suited for relational databases with some typical features:

- **Variable data:** semi-structured, evolving, or has no schema
- **Massive data:** terabytes or petabytes of data from new applications (web analysis, sensors, social graphs)
- **Parallelism:** large data requires architectures to handle massive parallelism, scalability, and reliability
- **Simpler queries:** may not need full SQL expressiveness
- **Relaxed consistency:** more tolerant of errors, delays, or inconsistent results ("eventual consistency")
- **Easier/cheaper:** less initial cost to get started

NoSQL is not really about SQL but instead developing data management systems that are not relational.

- NoSQL – "Not Only SQL"





## NoSQL Systems

There are a variety of systems that are not relational:

- MapReduce – useful for large scale analysis
- Key-value stores – ideal for retrieving specific data records from a large set of data
- Document stores – similar to key-value stores except value is a document in some form (e.g. JSON)
- Graph databases – represent data as graphs





## MapReduce

***MapReduce*** was invented by Google and has an open source implementation called Hadoop.

Data is stored in files. Users provide functions:

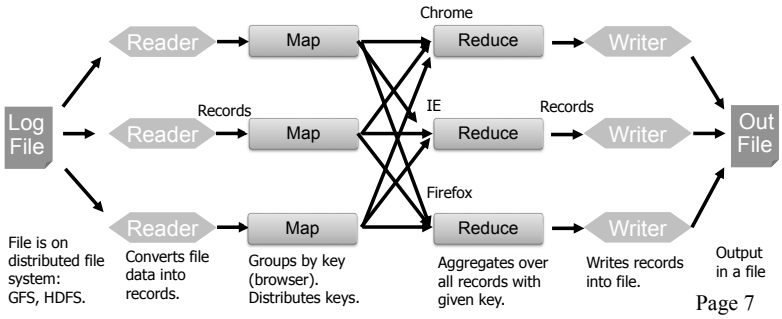
- reader(*file*) – converts file data into records
- map(*records*) – converts records into key-value pairs
- combine(*key, list of values*) – optional aggregation of pairs after map stage
- reduce(*key, list of values*) – summary on key values to produce output records
- write(file) – writes records to output file

MapReduce (Hadoop) provides infrastructure for tying everything together and distributing work across machines.

# MapReduce Example
# Web Data Analysis

Data file records: URL, timestamp, browser

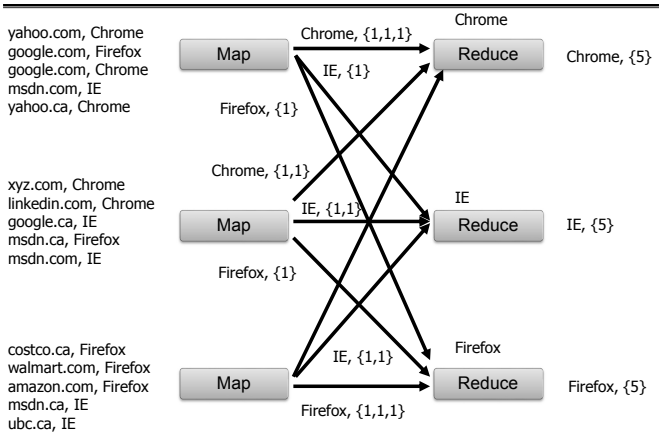
Goal: Determine the most popular browser used.








# MapReduce Example (2)
# Web Data Analysis








# MapReduce Extensions

The key benefit of MapReduce and Hadoop is their scalable performance, not that they do not support SQL. In fact, schemas and declarative SQL have many advantages.

Extensions to Hadoop combine the massive parallel processing with familiar SQL features:

- Hive – an SQL-like language variant
- Pig – similar to relational operators

Data manipulations expressed in these languages are then converted into a MapReduce workflow automatically.





# Key-Value Stores

***Key-value stores*** store and retrieve data using keys. The data values are arbitrary. Designed for "web sized" data sets.

Operations:

- insert(key, value)
- fetch(key)
- update(key)
- delete(key)

Benefits: high-scalability, availability, and performance

Limitations: single record transactions, eventual consistency, simple query interface

Systems: Cassandra, Amazon Dynamo, Google BigTable, HBase





# Document Stores

***Document stores*** are similar to key-value stores but the value stored is a structured document (e.g. JSON, XML).

Can store and query documents by key as well as retrieve and filter documents by their properties.

Benefits: high-scalability, availability, and performance

Limitations: same as key-value stores, may cause redundancy and more code to manipulate documents

Systems: CouchDB, MongoDB, SimpleDB

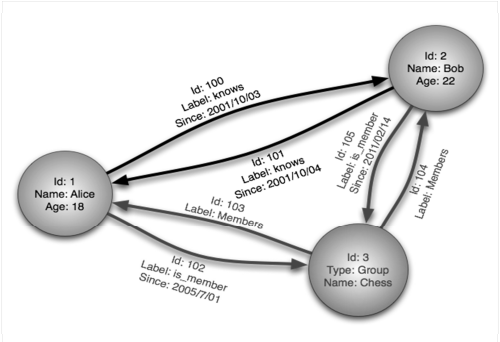




# Graph Databases

***Graph databases*** model data as nodes (with properties) and edges (with labels).

- Systems: Neo4J, FlockDB



## NoSQL Question

***Question:*** How many of the following statements are ***true***?

**1)** Neo4J is a document store.

**2)** Unlike Hadoop, a relational database is designed to restart any failed part of a query when a failure occurs.

**3)** Key-value stores are similar to a tree data structure.

**4)** SQL cannot be used to query MongoDB.

**5)** Relational systems typically scale better than NoSQL systems.

**A)** 0 **B)** 1 **C)** 2 **D)** 3 **E)** 4





## Conclusion

***NoSQL databases*** ("Not only SQL") is a category of data management systems that do not use the relational model.

There is a variety of NoSQL systems including:

- MapReduce systems
- Key-value stores
- Document stores
- Graph databases

NoSQL databases are designed for high performance, availability, and scalability at the compromise of restricted query languages and weaker consistency guarantees.





## Objectives

Understand alternative models for representing data besides the relational model.

List some NoSQL databases and reason about their benefits and issues compared to the relational model for certain problems.

Explain at a high-level how MapReduce works.

# COSC 304
# Introduction to Database Systems

## Triggers

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## General Enterprise Constraints Triggers

General enterprise constraints are often too complicated to be enforced using foreign keys and `CHECK`.

***Triggers*** are used to perform actions when a defined event occurs. They are standardized, but also have been present in various database systems for some time.

Triggers allow general constraints to be enforced in response to database events.





## Event-Condition-Action Rules

Triggers are also called ***event-condition-action*** (***ECA***) ***rules***.

When an ***event*** occurs (such as inserting a tuple) that satisfies a given ***condition*** (any SQL boolean-valued expression) an ***action*** is performed (which is any sequence of SQL statements).

Thus, triggers provide a very general way for detecting events and responding to them.





## Triggers Example

Consider this situation where triggers are useful.

The `WorksOn` relation has a foreign key to `Emp` (`eno`). If a user inserts a record in `WorksOn` and the employee does not exist, the insert fails.

However with triggers, we can accept the insertion into `WorksOn` and then create a new record in `Emp` so that the foreign key constraint is not violated.





## Example Trigger

```
CREATE TRIGGER insertWorksOn
  BEFORE INSERT ON WorksOn          <-- Event
  REFERENCING NEW ROW AS NewWO
  FOR EACH ROW                      <-- Condition
  WHEN (NewWO.eno NOT IN
          (SELECT eno FROM emp))
  INSERT INTO Emp (eno)
       VALUES (NewWO.eno);          <-- Action
```





## Triggers Standard Syntax

```
CREATE TRIGGER <name>
  BEFORE | AFTER | INSTEAD OF <events>
  [<referencing clause>]
  [FOR EACH ROW]
  [WHEN (<condition>)]
  <action>
```

Notes:

- `BEFORE`, `AFTER`, `INSTEAD OF` indicate *when* a trigger is executed.
- `<events>` is the events that the trigger will be executed for. It will be one of these events:

```
INSERT ON R
DELETE ON R
UPDATE [OF A1,A2,..,An] on R
```

## Triggers Syntax - *FOR EACH ROW*

There are two types of triggers:

- **row-level triggers** that are executed for each row that is updated, deleted, or inserted.
- **statement-level triggers** that are only executed once per statement regardless of how many tuples are affected.

Inserting the clause `FOR EACH ROW` indicates a row-level trigger (the default is a statement-level trigger).





## Triggers Syntax - Referencing

The referencing clause allows you to assign names to the row or table being affected by the triggered event:

- `INSERT` statements imply a new tuple (for row-level) or new set of tuples (for statement-level).
- `DELETE` implies an old tuple or table.
- `UPDATE` implies both.

These tuples or tables can be referred to using the syntax:

```
[NEW OLD] [TUPLE TABLE] AS <name>
```

Example: Statement-level trigger on an update:

```
REFERENCING OLD TABLE AS oldTbl
            NEW TABLE as newTbl
```





## Triggers Syntax - Condition

The condition is used to filter out when the trigger action is performed.

- For example, even though an insert event may occur, you may only want to execute the trigger if a certain condition holds.

Any SQL boolean-valued condition can be used. The condition clause is evaluated before or after the triggering event, depending on whether `BEFORE` or `AFTER` is used in the event.





## Triggers Syntax - Action

The action is performed when the event occurs and the condition is satisfied.

The action is one or more SQL statements.

If more than one SQL statement is executed, they should be in a `BEGIN .. END` block.





## Triggers Example #2

Consider a trigger that is used to monitor employee salary increases.

If an employee's salary is increased by more than 10%, we want to add that employee to a relation `auditEmp(eno,newSalary,oldSalary)` that is used by the accounting department to verify that the employee is not stealing from the company.





## Example Trigger #2

```
CREATE TRIGGER cheatingEmployee
  AFTER UPDATE OF salary on Emp
  REFERENCING
     OLD ROW AS old
     NEW ROW AS new
  FOR EACH ROW
  WHEN (new.salary > old.salary*1.1)
   INSERT INTO auditEmp
     VALUES (new.eno,new.salary,old.salary);
```

Event - only fired when salary updated

UPDATE - have old and new tuples

One tuple at a time

Condition

Action - add new tuple to auditEmp

## Triggers and Views

One interesting use of triggers is to allow updates on views that are not normally updatable.

This is done by using the `INSTEAD OF` clause. When an update is executed, instead of executing the update command given by the user, the trigger executes its own SQL statements.

For instance, this may allow it to perform inserts on many tables that were combined to form a single view.





## Triggers Question

***Question:*** **True or False:** A standard trigger can be used to perform an action in response to a SELECT query.

**A)** true

**B)** false





## Triggers In Practice

The syntax presented is for the SQL standard. Each database has its own syntax and level of support for triggers.

| Feature | MySQL | SQL Server | PostgreSQL |
|---|---|---|---|
| Statement-level triggers | No | Yes | Yes |
| Row-level triggers | Yes | No | Yes |
| OLD/NEW Table | No | Yes. Inserted and deleted tables. | No |
| OLD/NEW Row | Yes. Use New and Old as names. | No | Yes. Use New and Old as names. |
| INSTEAD OF trigger | No | Yes, with limits | Yes, only on views |
| WHEN clause | No | | Yes |



## Example Trigger in MySQL Syntax

```
CREATE TRIGGER cheatingEmployee
   AFTER UPDATE ON Emp
   FOR EACH ROW
   BEGIN
     IF NEW.salary > OLD.salary*1.1 THEN
       INSERT INTO auditEmp
        VALUES (NEW.eno,NEW.salary,OLD.salary);
     END IF;
    END;
```





## Example Trigger in SQL Server Syntax

```
CREATE TRIGGER cheatingEmployee ON Emp
  AFTER UPDATE AS
  BEGIN
    INSERT INTO auditEmp
      SELECT d.eno, newsal, oldsal FROM
         deleted d JOIN inserted i ON
         d.eno = i.eno
         WHERE i.salary > d.salary*1.1
  END;
```





## Triggers Practice Questions

Write triggers for these cases:

- 1) When an employee is added to the `Emp` table in `Department` '`D1`' with no supervisor make their supervisor '`E8`'.
- 2) Write the same trigger as a statement-level trigger.
- 3) Write a trigger that deletes all tuples in the `WorksOn` relation for an employee when the employee is deleted from the `Emp` relation.

## Conclusion

More complicated constraints can be enforced in the database using triggers.

Triggers allow for efficient enforcement of general constraints by specifying specific events to trap and actions to take.





## Objectives

- Create triggers from high-level descriptions
- Explain the difference between row level and statement level triggers
- List the events that triggers are used for
- Explain how triggers can be used for views

# COSC 304
# Introduction to Database Systems

## Views and Security

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Views

A ***view*** is a named query that is defined in the database.

- To the user, a view appears just like any other table and can be present in any SQL query where a table is present.

A view may either be:

- *virtual* - produced by a SQL query on demand.
- *materialized* - the view is stored as a derived table that is updated when the tables that it refers to are updated.





## Creating Views

Views are created using the **`CREATE VIEW`** command:

```
CREATE VIEW viewName [(col1,col2,...,colN)]
AS selectStatement [WITH CHECK OPTION]
```

Notes:

- Select statement can be any SQL query and is called the ***defining query*** of the view.
- It is possible to rename the columns when defining the view, otherwise they default to the names produced by the query.
- `WITH CHECK OPTION` is used to insure that if updates are performed through the view, the updated rows must satisfy the `WHERE` clause of the query.





## Views Example

Create a view that has only the employees of department 'D2':

```
CREATE VIEW empD2
AS SELECT * FROM emp WHERE dno = 'D2';
```

Create a view that only shows the employee number, title, and name:

```
CREATE VIEW staff (Number,Name,Title)
AS SELECT eno,ename,title FROM emp;
```

- The first example is a *horizontal view* because it only contains a subset of the rows.
- The second example is a *vertical view* because it only contains a subset of the columns.





## Removing Views

Views are removed using the **`DROP VIEW`** command:

```
DROP VIEW viewName [RESTRICT|CASCADE]
```

Notes:

- `RESTRICT` will not delete a view if other views are dependent on it.
- `CASCADE` deletes the view and all dependent views.





## View Resolution

When a query uses a view, the process of ***view resolution*** is performed to translate the query into a query over only the base relations.

```
CREATE VIEW staff (Number,Name,Job)
AS SELECT eno,ename,title FROM emp WHERE dno = 'D2';

SELECT Number, Name FROM staff
WHERE job = 'EE' ORDER BY Number;
```

Step #1: Replace column names in `SELECT` clause with names in defining query.

```
SELECT eno, ename FROM staff
WHERE job = 'EE' ORDER BY Number;
```

## View Resolution (2)

Step #2: View names in FROM clause replaced by FROM clause in defining query.

```
SELECT eno, ename FROM emp
WHERE job = 'EE' ORDER BY Number;
```

Step #3: WHERE clause of user and defining query are combined. Replace column view names with names in defining query.

```
SELECT eno, ename FROM emp
WHERE title = 'EE' AND dno = 'D2' ORDER BY Number;
```





## View Resolution (3)

Step #4: GROUP BY and HAVING clause copied to user query from defining query. (no change in example)

```
SELECT eno, ename FROM emp
WHERE title = 'EE' AND dno = 'D2' ORDER BY Number;
```

Step #5: Rename fields in ORDER BY clause.

```
SELECT eno, ename FROM emp
WHERE title = 'EE' AND dno = 'D2' ORDER BY eno;
```





## View Limitations

1) If a view contains an aggregate function, it can only be used in SELECT and ORDER BY clauses of queries.

2) A view whose defining query has a GROUP BY cannot be joined with another table or view.





## View Updatability

When a *base table is updated*, all views defined over that base table are automatically updated. When an *update is performed on the view*, it is possible to update the underlying table.

A view is only *updatable* if:

- does not use DISTINCT
- every element in SELECT is a column name (no derived fields)
- contains all fields of base relation that are non-NULL
- FROM clause contains only one table
- WHERE does not contain any nested selects
- no GROUP BY or HAVING in defining query

In practice, a view is only updatable if it uses only a single table, and any update through the view must not violate any of the integrity constraints of the table.





## View *WITH CHECK OPTION*

**WITH CHECK OPTION** is used for updatable views. It is used to prevent updates through a view that would then remove rows from the view.

```
CREATE VIEW staff (Number,Name,Job,DeptNum)
AS SELECT eno,ename,title,'D2' FROM emp
                  WHERE DeptNum = 'D2';

UPDATE staff SET DeptNum = 'D3' WHERE Number='E3';
```

This update would remove the employee E3 from the view because changing his department to 'D3' no longer matches the view which only contains employees in department 'D2'. To prevent this, can specify the WITH CHECK OPTION.

```
CREATE VIEW staff (Number,Name,Job,DeptNum)
AS SELECT eno,ename,title FROM emp WHERE DeptNum = 'D2'
WITH CHECK OPTION;
```





## View Updatable Question

***Question:*** Select one view definition that would be updatable.

**A)**
```
CREATE VIEW v AS
SELECT DISTINCT salary FROM emp
```

**B)**
```
CREATE VIEW v AS
SELECT * FROM emp WHERE eno > 'E3'
```

**C)**
```
CREATE VIEW v AS
SELECT name FROM emp
```

**D)**
```
CREATE VIEW v AS
SELECT eno, salary/12 FROM emp
```

## Advantages and Disadvantages of Views

Advantages:

- Data independence - allows base relations to change without affecting users.
- Security - views can be used to limit access to certain data to certain users.
- Easier querying - using views can often reduce the complexity of some queries.
- Convenience/Customization - users only see the parts of the database that they have interest and access to.

Disadvantages:

- Updatable views are not always supported and are restrictive.
- Performance - views add increased overhead: during query parsing and especially if they are materialized.





## Views Practice Questions

Creates views that:

- 1) Show only employees that have title '`EE`'.
- 2) List only projects that have someone working on them.
- 3) List only the employees that manage another employee.
- 4) List the department number, name, and average employee salary in the department.
- 5) Show all employees but only lists their number, name, and title.
- 6) Show all employee and `worksOn` information for employee '`E1`'.





## SQL Security

Security in SQL is based on ***authorization identifiers***, ***ownership***, and ***privileges***.

An authorization identifier (or user id) is associated with each user. Normally, a password is also associated with a authorization identifier. Every SQL statement performed by the DBMS is executed on behalf of some user.

The authorization identifier is used to determine which database objects the user has access to.

Whenever a user creates an object, the user is the owner of the object and initially is the only one with the ability to access it.





## SQL Privileges

***Privileges*** give users the right to perform operations on database objects. The set of privileges are:

- `SELECT` - the user can retrieve data from table
- `INSERT` - the user can insert data into table
- `UPDATE` - the user can modify data in the table
- `DELETE` - the user can delete data (rows) from the table
- `REFERENCES` - the ability to reference columns of a named table in integrity constraints
- `USAGE` - the ability to use domains, character sets, and translations (i.e. other database objects besides tables)

Notes:

- `INSERT` and `UPDATE` can be restricted to certain columns.
- When a user creates a table, they become the owner and have full privileges on the table.





## SQL *GRANT* Command

The **`GRANT`** command is use to give privileges on database objects to users.

```
GRANT {privilegeList | ALL [PRIVILEGES]}
  ON ObjectName
  TO {AuthorizationIdList | PUBLIC}
  [WITH GRANT OPTION]
```

The privilege list is one or more of the following privileges:

```
SELECT [(columnName [,...]]
DELETE
INSERT [(columnName [,...]]
UPDATE [(columnName [,...]]
REFERENCES [(columnName [,...]]
USAGE
```





## SQL *GRANT* Command (2)

The `ALL PRIVILEGES` keyword grants all privileges to the user except the ability to grant privileges to other users.

The `PUBLIC` keyword grants access to all users (present and future) of the database.

The `WITH GRANT OPTION` allows users to grant privileges to other users. A user can only grant privileges that they themselves hold.

## *GRANT* Examples

Allow all users to query the `Dept` relation:

```
GRANT SELECT ON dept TO PUBLIC;
```

Only allow users `Manager` and `Director` to access and change `Salary` in `Emp`:

```
GRANT SELECT, UPDATE(salary) ON Emp TO Manager,Director;
```

Allow the `Director` full access to `Proj` and the ability to grant privileges to other users:

```
GRANT ALL PRIVILEGES ON Proj TO Director
    WITH GRANT OPTION;
```





## Required Privileges Example

What privileges are required for these statements:

```
UPDATE Emp SET salary=salary*1.1 WHERE eno IN (
    SELECT eno FROM WorksOn WHERE hours > 30);
```

Answer:

`SELECT on Emp`

`UPDATE(salary) on Emp`

`SELECT on WorksOn`





## Required Privileges Question

***Question:*** What are the required privileges for this statement?

```
DELETE FROM dept WHERE dno NOT IN
    (SELECT dno FROM WorksOn);
```

**A)** `DELETE, SELECT`

**B)** `DELETE on dept, DELETE on WorksOn`

**C)** `DELETE on dept, SELECT on WorksOn`

**D)** `DELETE on dept`

**E)** `DELETE on dept, SELECT on WorksOn, SELECT on dept`





## Required Privileges Question (2)

***Question:*** What are the required privileges for this statement?

```
INSERT INTO WorksOn (eno,pno)
            VALUES ('E5','P5');
```

**A)** `INSERT on WorksOn`

**B)** `INSERT`

**C)** `INSERT, SELECT`

**D)** `INSERT on WorksOn, UPDATE on WorksOn`

**E)** `none`





## GRANT Question

***Question:*** **True or False:** A user `WITH GRANT OPTION` can grant a privilege that they do not hold themselves.

**A)** true

**B)** false





## GRANT Question (2)

***Question:*** **True or False:** A users may be granted the same privilege on the same object from multiple users.

**A)** true

**B)** false

## SQL REVOKE Command

The REVOKE command is used to remove privileges on database objects from users.

```
REVOKE [GRANT OPTION FOR] {privilegeList | ALL [PRIVILEGES]}
  ON ObjectName
  FROM {AuthorizationIdList | PUBLIC} {RESTRICT|CASCADE}
```

Notes:

- ALL PRIVILEGES removes all privileges on object.
- GRANT OPTION FOR removes the ability for users to pass privileges on to other users (not the privileges themselves).
- RESTRICT - REVOKE fails if privilege has been passed to other users.
- CASCADE - removes any privileges and objects created using the revoked privileges including those passed on to others.





## SQL REVOKE Command (2)

Privileges are granted to an object to a user *from* a specific user. If a user then revokes their granting of privileges, that only applies to that user.

Example:

- User A grants all privileges to user B on table T.
- User B grants all privileges to user C on table T.
- User E grants SELECT privilege to user C on table T.
- User C grants all privileges to user D on table T.
- User A revokes all privileges on table T from B (using cascade).
- This causes all privileges to be removed for user C as well, except the SELECT privilege which was granted by user E.
- User D now has only the SELECT privilege as well.





## REVOKE Examples

Allow no users to query the Dept relation:

```
REVOKE SELECT ON dept FROM PUBLIC;
```

Remove all privileges from user Joe on Emp table:

```
REVOKE ALL PRIVILEGES ON Emp FROM Joe;
```





## REVOKE Question

***Question:*** User *A* executes:

```
GRANT SELECT, UPDATE ON T TO B WITH GRANT OPTION;
```

then User B executes:

```
GRANT SELECT, UPDATE ON T TO C;
```

then User A executes:

```
REVOKE GRANT OPTION FOR SELECT, UPDATE ON T FROM B;
```

**True or False:** User *C* loses SELECT on *T*.

**A)** true
**B)** false





## SQL Security and Views

Views are used to provide security and access restriction to certain database objects.

Example: Consider the Emp relation. We want the user Staff to have only query access but not be able to see users birthdate and salary. How do we accomplish this?

Step #1: Create a view on the Emp relation.

```
CREATE VIEW EmpView AS
SELECT eno, ename, title, supereno, dno FROM emp;
```





## SQL Security and Views (2)

Step #2: Provide SELECT privilege to staff.

```
GRANT SELECT ON EmpView TO Staff;
```

Step #3: REVOKE privileges on base relation (Emp)

```
REVOKE SELECT ON Emp From Staff;
```

## Privileges on Views Question

***Question:*** Table *T* is part of view definition query for view *V*. If user *A* has `SELECT` on *V*, but not on *T*, can user *A* run a query on *V* and see results?

**A)** yes

**B)** no





## Security Practice Questions

Use `GRANT` and `REVOKE` to provide the desired access for each of these cases. You may also need views. Assume that you are the DBA who has full privileges (owner) of all objects currently in the database.

1) Allow all users access to the `Dept` relation.

2) Allow the user `Accounting` to read and update the `salary` of `Emp`.

3) Allow user `Davis` to have full control of the employees in Dept `D2` including the ability to grant privileges to others.

4) Allow user `Smith` to only see their employee record (`eno='E3'`) and `WorksOn` information.

5) `Davis` is replaced as head of Dept. `D2`, so only leave him query access to employees in `D2`.





## Conclusion

Views are used to define virtual relations over base relations. This may be useful to:

- reduce query complexity
- improve performance (especially if view is materialized)
- provide different user views of the database
- allow security to be enforced on subsets of the schema

SQL security is enforced using `GRANT`/`REVOKE`.

- Views are useful for security as users can be given privileges to access a view but not the entire relation.





## Objectives

Views:

- define views using `CREATE VIEW` from high-level description
- explain and illustrate the process of view resolution
- explain when a view is updateable and argue why updating views is not possible in certain cases
- explain what the `WITH CHECK OPTION` does for views
- list advantages and disadvantages of views

Security:

- be able to use `GRANT`/`REVOKE` syntax
- list the types of privileges and know when to use them
- given a SQL command, explain what privileges are required for it to execute
- explain how views are useful for security

# COSC 304
# Introduction to Database Systems

## Advanced SQL

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Transaction Management Overview

The database system must ensure that the data stored in the database is always ***consistent***. There are two challenges in preserving database consistency:

- 1) The database system must handle ***failures*** of various kinds such as hardware failures and system crashes.
- 2) The database system must support ***concurrent execution*** of multiple transactions and guarantee that this concurrency does not lead to inconsistency.

A ***transaction*** is an ***atomic*** program that executes on the database and preserves the consistency of the database.

- The input to a transaction is a consistent database AND the output of the transaction must also be a consistent database.
- A transaction must execute completely or not at all.





## Transaction Management Motivating Example

Consider a person who wants to transfer $50 from a savings account with balance $1000 to a checking account with current balance = $250.

- 1) At the ATM, the person starts the process by telling the bank to remove $50 from the savings account.
- 2) The $50 is removed from the savings account by the bank.
- 3) Before the customer can tell the ATM to deposit the $50 in the checking account, the ATM "crashes."

Where has the $50 gone?

It is lost if the ATM did not support transactions!
The customer wanted the withdraw and deposit to both happen in one step, or neither action to happen.





## ACID Properties

To preserve integrity, transactions have the following properties:

- ***Atomicity*** - Either all operations of the transaction are properly reflected in the database or none are.
- ***Consistency*** - Execution of a transaction in isolation preserves the consistency of the database.
- ***Isolation*** - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- ***Durability*** - After a transaction successfully completes, the changes it has made to the database persist, even if there are system failures.





## ACID Properties

***Question:*** Two transactions running at the same time can see each other's updates. What ACID property is violated?

**A)** atomicity
**B)** consistency
**C)** isolation
**D)** durability
**E)** none of them





## Transaction Definition in SQL

In SQL, a transaction begins implicitly.

A transaction in SQL ends by:

- **Commit** accepts updates of current transaction.
- **Rollback** aborts current transaction and discards its updates. Failures may also cause a transaction to be aborted.

An ***isolation level*** reflects how a transaction perceives the results of other transactions. It applies only to your perspective of the database, not other transactions/users. Lowering isolation level improves performance but may potentially sacrifice consistency.

## Example Transactions

Transaction to deposit $50 into a bank account:

```
-- TRANSACTION T1:
UPDATE Account WHERE num = 'S1' SET balance=balance+50;
COMMIT;
```

Transaction to calculate totals for all accounts (twice):

```
-- TRANSACTION T2:
SELECT SUM(balance) as total1 FROM Account;
SELECT SUM(balance) as total2 FROM Account;
COMMIT;
```

Transaction to add a new account:

```
-- TRANSACTION T3:
INSERT INTO ACCOUNT (num, balance) VALUES ('S5', 100);
COMMIT;
```





## Levels of Consistency in SQL

The isolation level can be specified by:

`SET TRANSACTION ISOLATION LEVEL =` *X* where *X* is

- **Serializable -** transactions behave like executed one at a time.
- **Repeatable read -** repeated reads must return same data. Does not necessarily read newly inserted records.
- **Read committed -** only committed values can be read, but successive reads may return different values.
- **Read uncommitted** - even uncommitted records may be read. Reading an uncommitted value is called a *dirty read*.





## Scheduling of Transactions

Each transaction in a database is a separate executing program.

- A transaction may be its own program or a thread of execution.

The operating system schedules the execution of programs outside of the control of the DBMS.

- Thus, transactions may be executed in any order (as long as the order of operations within a transaction are the same). This interleaving is what produces different schedules.

The DBMS uses its concurrency control protocol to restrict the schedules to those that respect the consistency specified by the user for the transaction isolation level.

- All transactions must write lock any data item updated and the relation lock if inserting.
- Isolation level only affects read locks.





## Transaction Schedule Example

Transaction **`T2`** that does two queries has its statements interleaved with transaction **`T1`** that does an update:

```
SELECT SUM(balance) as total1 FROM Account;  --T2
UPDATE Account WHERE num='S1' SET balance=balance+50;--T1
COMMIT;  -- T1
SELECT SUM(balance) as total2 FROM Account; -- T2
COMMIT; -- T2
```

With isolation level ***read committed***, `total1` will not be the same as `total2`.

With isolation level ***repeatable read***, this schedule is not possible as **`T2`** will lock all accounts stopping **`T1`** from updating.





## Transaction Schedule Example (2)

Transaction **`T2`** that does two queries has its statements interleaved with transaction **`T3`** that does an insert:

```
SELECT SUM(balance) as total1 FROM Account;  --T2
INSERT INTO ACCOUNT (num, balance) VALUES ('S5' , 100);
COMMIT;  -- T3
SELECT SUM(balance) as total2 FROM Account; -- T2
COMMIT; -- T2
```

With isolation level ***repeatable read***, `total1` will not be the same as `total2`.

With isolation level ***serializable***, this schedule is not possible as **`T2`** will lock the account table, stopping **`T3`** from inserting.





## Summary of Isolation Levels

| Isolation Level | Problems | Lock Usage | Speed | Comments |
|---|---|---|---|---|
| Serializable | None | Read locks held to commit ; read lock on relation | Slowest | Only level that guarantees correctness. |
| Repeatable read | Phantom tuples | Read locks held to commit | Medium | Useful for modify transactions. May not see tuples inserted by others. |
| Read committed | Phantom tuples, values may change | Read locks released after each statement | Fast | Useful for transactions where operations are separable but updates are all or none.<br>Re-reading same value may produce different results. |
| Read uncommitted | Phantoms, values may change, dirty reads | No read locks | Fastest | Useful for read-only transactions that tolerate inaccurate results. May see updates that will never be committed. |

## Read Committed Question

***Question:*** Will this transaction always see the same results for both queries if executed using **READ COMMITTED**?

```
SELECT SUM(balance) as total1 FROM Account;
SELECT SUM(balance) as total2 FROM Account;
COMMIT;
```

**A)** yes

**B)** no





## Repeatable Read Question

***Question:*** Will this transaction always see the same results for both queries if executed using **REPEATABLE READ**?

```
SELECT SUM(balance) as total1 FROM Account;
SELECT SUM(balance) as total2 FROM Account;
COMMIT;
```

**A)** yes

**B)** no





## JDBC Transaction Example

```
try (Connection con = DriverManager.getConnection(url, uid, pw);
     Statement stmt = con.createStatement();)
{
   con.setAutocommit(false);      ← Force explicit commit/rollback
   ResultSet rst = stmt.executeQuery("SELECT ename,salary
                                      FROM Emp WHERE eno='E1'");

   if (rst.next() && rst.getDouble(2) < 50000)
   {
       stmt.executeUpdate("UPDATE Emp SET salary=100000
                           WHERE eno='E1'");
       con.commit();              ← Commit work to DB
   }
   con.rollback();                ← Rollback work done
}
catch (SQLException ex)
{
   System.err.println(ex); con.rollback();
}
```





## Advanced SQL Overview

The SQL standard has evolved to define a language that is computationally complete and supports features required by a modern object-relational DBMS.

An ***object-relational DBMS*** is a relational DBMS extended to support object-oriented features such as inheritance, user defined types, and polymorphism.

- Supports active code such as triggers and stored procedures.
- Handle impedance mismatch between program code and SQL.
  - ⇨SQL is declarative and set based -- Java is procedural and object based.
- Extend database to support new types and operations.





## User-Defined Types in SQL

General form (simple version):

```
CREATE TYPE typeName AS builtInType [FINAL];
```

⇨`Final` means that cannot create subtypes of the new type.

Example: Create `SSN` type:

```
CREATE TYPE SSN AS VARCHAR(10) FINAL;
```

User-defined types may either extend base types as shown here or define a structure similar to a class with attributes, routines, and operators.





## User-Defined Types Example

```
CREATE TYPE PersonType AS (
   dateOfBirth DATE,
   fName       VARCHAR(15),
   lName       VARCHAR(15),
   sex         CHAR)
INSTANTIABLE                                  ← Can create objects
NOT FINAL                                     ← Can create subtypes
REF IS SYSTEM GENERATED                       ← DB creates references
INSTANCE METHOD age() RETURNS INTEGER;        ← Define method

CREATE INSTANCE METHOD age() RETURNS INTEGER
   FOR PersonType
   BEGIN
      RETURN /* Code to find age from Self.dateOfBirth */
   END;
```

## Subtypes in SQL

General form:

```
CREATE TYPE typeName UNDER inheritType AS ...
```

Example:

```
CREATE TYPE StaffType UNDER PersonType AS (
   staffNo     VARCHAR(5),
   salary      DECIMAL(7,2))
   INSTANTIABLE
   NOT FINAL;
```

A subtype inherits all methods and attributes from its supertype. It may override any inherited methods/attributes.





## Recursive Queries in SQL

General form:

```
WITH RECURSIVE tableName(attr1,attr2,..attrN) AS
   <SELECT query that defines recursive relation>
<SELECT query that uses recursive relation>
```

Example: Return all employees supervised by 'J. Jones'.

```
WITH RECURSIVE supervises(supId,empId) AS
   (SELECT supereno, eno FROM emp)
   UNION
   (SELECT S1.supId, S2.empId
   FROM supervises S1, supervises S2
   WHERE S1.empId = S2.supId)
SELECT E1.ename FROM supervises, emp AS E, emp AS E2
WHERE supervises.supId = E2.eno and E2.ename = 'J. Jones'
       and supervises.empId = E1.ename;
```





## Conclusion

In SQL, different isolation levels can be specified:

- serializable, repeatable read, read committed, read uncommitted
- Weaker forms of isolation do not guarantee the ACID properties, but may be useful for read transactions that do not need exact data and require faster execution.

Object-relational DBMSs support user defined data types, active consistency checks (triggers), inheritance, and recursive queries.





## Objectives

- List and explain the isolation levels in SQL.
- List some features of object-relational DBMSs.

# COSC 304
# Introduction to Database Systems

## XML

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## XML

***Extensible Markup Language*** (***XML***) is a markup language that allows for the description of data semantics.

- XML is a markup language for describing any type of data because the markup terms can be user-defined.
- XML is **case-sensitive** unlike HTML.

XML is a standard by the World Wide Web Consortium (W3C).





## Advantages of XML

Some advantages of XML:

- Simplicity
  - ⇨The XML standard is relatively short and simple.
- Open standard
  - ⇨Standardized by W3C to be vendor and platform independent.
- Extensibility
  - ⇨XML allows users to define their own tags.
- Separation of data and presentation
  - ⇨XML data may be presented to the user in multiple ways.
- Interoperability
  - ⇨By standardizing on tags, interoperation and integration of systems is simplified.





## XML Components

An XML document is a text document that contains markup in the form of ***tags***.

An XML document consists of:

- An XML *declaration line* indicating the XML version.
- *Elements* (or tags) called *markup*. Each element may contain free-text, attributes, or other nested elements.
  - ⇨Every XML document has a single root element.
  - ⇨Tags, as in HTML, are matched pairs, as <foo> … </foo>.
  - ⇨Closing tags are not needed if the element contains no data: <foo/>
  - ⇨Tags may be nested.
- An *attribute* is a name-value pair declared inside an element.
- Comments

XML data is ordered by nature.





## XML Example

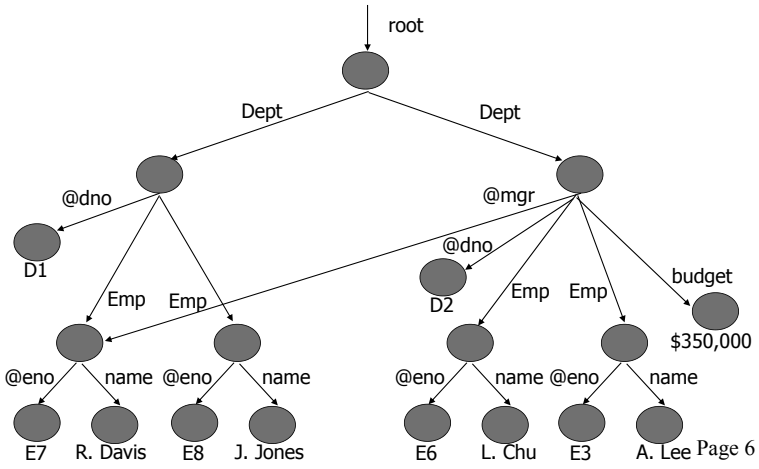
```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<?xml:stylesheet type="text/xsl" href="dept.xsl"?>
<!DOCTYPE root SYSTEM "dept.dtd">

<!-- Department/Employee data formatted in XML -->

<root>
  <Dept dno = "D1">
        <Emp eno="E7"><name>R. Davis</name></Emp>
        <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
        <Emp eno="E6"><name>L. Chu</name></Emp>
        <Emp eno="E3"><name>A. Lee</name></Emp>
        <budget>350000</budget>
  </Dept>
</root>
```

XML declaration
Stylesheet for presentation
External DTD for validation
Comment
Attribute
Element reference





## XML (tree view)



## Well-Formed and Valid XML Documents

An XML document is ***well-formed*** if it obeys the syntax of the XML standard. This includes:

- Having a single root element
- All elements must be properly closed and nested.

An XML document is ***valid*** if it is well-formed and it conforms to a Document Type Definition (DTD) or an XML Schema Definition (XSD).

- A document can be well-formed without being valid if it contains tags or nesting structures that are not allowed in its DTD/XSD.
- The DTD/XSD are schema definitions for an XML document.





## XML Well-Formed Question

***Question:*** How many of these two documents are well-formed?

**1:** `<x><a>Test</a><bT></bt></x>`

**2:** `<x>abc</x><y>def</y>`

**A)** 0
**B)** 1
**C)** 2





## Namespaces

***Namespaces*** allow tag names to be qualified to avoid naming conflicts. A naming conflict would occur when the same name is used by two different domains or vocabularies.

A namespace consists of two components:

- 1) A declaration of the namespace and its abbreviation.
- 2) Prefixing tag names with the namespace name to exactly define the tag's origin.





## Namespaces Example

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<root xmlns = "http://www.foo.com"          ← Default namespace
      xmlns:n1 = "http://www.abc.com">      ← n1 namespace
  <Dept dno = "D1">
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <n1:budget>350000</budget>
  </Dept>                  ← budget is a XML tag in the n1 namespace.
</root>
```





## Schemas for XML

Although an unrestricted XML format is useful to some applications, database data normally has some structure, even though that structure may not be as rigid as relational schemas.

It is valuable to define schemas for XML documents that restrict the format of those documents.

There are two main ways of specifying a schema for XML:

- Document Type Definition (DTD) (original, older)
- XML Schema





## Document Type Definitions (DTDs)

A ***Document Type Definition*** (***DTD***) defines the grammatical rules for the document. It is not required for an XML document but provides a mechanism for checking a document's validity.

General DTD form:

```
<!DOCTYPE myroot [ <elements> ]>
```

name of root element in XML document

contents of DTD declares elements and attributes

Essentially, a DTD is a set of document rules expressed using EBNF (Extended Backus-Naur Form) grammar. The rules limit:

- the set of allowable element names, how elements can be nested, and the attributes of an element among other things

## DTD Example

```
<!DOCTYPE root [
  <!ELEMENT root(Dept+)>
  <!ELEMENT Dept(Emp*, budget?)>
      <!ATTLIST Dept dno ID #REQUIRED>
      <!ATTLIST Dept mgr IDREF #IMPLIED>
  <!ELEMENT budget (#PCDATA)>
  <!ELEMENT Emp (name)>
      <!ATTLIST Emp eno ID #REQUIRED>
  <!ELEMENT name (#PCDATA)>
]>
```

+ means 1 or more times

* means 0 or more times

? means 0 or 1 time

Element reference (like a foreign key)

ID is a unique value that identifies the element

Parsed Character Data (atomic value)





## DTD Elements

Each element declaration has the form:

`<!ELEMENT <name> ( <components> )>`

Notes:

- Each `!ELEMENT` declaration specifies that an element is being created.
- Components is either a list of one or more subelements or `#PCDATA`, or `EMPTY` (has no content).
- Each subelement may occur exactly once, zero or one time (`?`), zero or more (`*`), or one or more (`+`).
- An element that is a composite element (contains other elements) specifies the other elements in parentheses. These elements must appear in the document in the order specified.





## DTD Attributes

Attributes are declared using:

`<!ATTLIST <eltName> <attrName> <type> <value>)`

Notes:

- Attribute names must be unique within an element.
- The type of attribute may be (among others):
  - `CDATA` - non-parsed string
  - `ID` - element identifier (`ID` valued attributes must start with a letter)
  - `IDREF` or `IDREFS` - one or more element references
- The value source may be:
  - `#IMPLIED` - attribute is optional
  - `#REQUIRED` - always present
  - `#FIXED "default value"`- declared default value





## DTD Attribute Example

Values for an attribute may be specified:

```
<!ELEMENT Emp (name,salary)>
<!ATTLIST Emp title ("EE"|"SA"|"PR"|"ME") "EE">
<!ATTLIST Emp eno ID #REQUIRED>
<!ATTLIST Emp bdate CDATA #IMPLIED>
```





## DTD *ID* and *IDREF*

An `ID` attribute is used to uniquely identify an element within a document. It is used to model keys.

- Note however that the scope of the uniqueness is the entire document not elements of the given type.
- An `ID` is like declaring named anchors in HTML.
- An ID must be a string starting with a letter and must be unique throughout the ***whole*** document.

`IDREF`/`IDREFS` are attributes that allow an element to refer to another element in the document.

- An `IDREF` attribute contains a single value that references a named location in the document (an existing `ID` value).
- An `IDREFS` attribute contains a list of `ID` references.
- Using `IDREF` allows the structure of an XML document to be a graph, rather than just a tree.





## DTD Choice

The symbol "`|`" can connect alternative sequences of tags.

For example, a name may have an optional title, a first name, and a last name, in that order, or is a full name:

```
<!ELEMENT NAME (
  (TITLE?, FIRST, LAST) | FULLNAME
)>
```

## Using DTDs in an XML Document

A DTD can be either embedded in the XML document itself or specified as an external file.

Embedding:

- Set `STANDALONE = "yes"` in XML declaration.
- Put DTD right after XML declaration line.

Separate file:

- Set `STANDALONE = "no"` in XML declaration.
- Put DTD in another file say `myfile.dtd`.
- Put the line in the XML document:

```
<!DOCTYPE myroot SYSTEM "myfile.dtd">
```

name of root element in XML document | private DTD | DTD file location





## DTD Question

Build a DTD for our relational database:

- Emp (eno, ename, title, salary, dno)
- Project (pno, pname, budget, dno)
- WorksOn (eno, pno, resp, dur)
- Department (dno, dname, mgr)





## XML Schema

While DTDs are sufficient for many uses of XML, they are lacking compared to other forms of database schema. Specifically, they do not have good support for:

- data types
- constraints
- explicit primary keys and foreign key references

Further, DTDs are written in a non-XML syntax.

XML Schema was defined by the W3C to provide a standard XML schema language which itself is written in XML and has better support for data modeling.





## Types in XML Schema

Elements that contain other elements are called ***complex types***:

```
<xsd:element name = "Dept">
  <xsd:complexType>
       <xsd:sequence>
             // Children defined here
       </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Elements that have no subelements are ***simple types***:

```
<xsd:element name = "name" type = "xsd:string" />
<xsd:element name = "budget" type = "xsd:decimal" />
```





## Cardinality in XML Schema

The number of occurrences of elements can be controlled by cardinality constraints `minOccurs` and `maxOccurs`:

```
<xsd:element name = "Emp"
            minOccurs="0" maxOccurs="unbounded">

<xsd:element name = "budget" type = "xsd:decimal"
  minOccurs="1" maxOccurs="1" />
```





## Constraints in XML Schema

***Uniqueness constraints*** dictate that the value of an element (with a specified path) must be unique:

```
<xsd:unique name = "UniqueDno">
  <xsd:selector xpath = "Dept" />
  <xsd:field xpath = "@dno" />
</xsd:unique>
```

***Key constraints*** are uniqueness constraints where the value cannot be null.

```
<xsd:key name = "EmpKey">
  <xsd:selector xpath = "Dept/Emp" />
  <xsd:field xpath = "@eno" />
</xsd:key>
```

## Constraints in XML Schema (2)

***Reference constraints*** forces the value of a reference to be constrained to specified keys:

```
<xsd:keyref name = "DeptMgrFK" refer="EmpKey">
  <xsd:selector xpath = "Dept" />
  <xsd:field xpath = "@mgr" />
</xsd:keyref>
```





## Other Features of XML Schema

XML Schema also allows you to:

- Define your own types
- Define schema references to simplify schema maintenance
- Define groups of elements or attributes
- Use groups to allow for schema variations and choices
- Construct elements that are lists or unions of values





## XML Schema Example

```
<?xml version = "1.0">
<xsd:schema xmlns:xsd = "http://www.w3.org/2000/10/XMLSchema">
<xsd:element name = "root">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "Dept" minOccurs="1" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:attribute name = "dno" type = "xsd:string" />
          <xsd:attribute name = "mgr" type = "xsd:string" />
          <xsd:element name = "Emp" minOccurs="0" maxOccurs="unbounded">
            <xsd:complexType>
              <xsd:element name = "name" type = "xsd:string" />
              <xsd:attribute name = "eno" type = "xsd:string" />
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="budget" minOccurs="0" type ="xsd:decimal" />
        </xsd:complexType>
      </xsd:element>
```





## XML Schema Example

```
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:key name = "DeptKey">
  <xsd:selector xpath = "Dept" />
  <xsd:field xpath = "@dno" />
</xsd:key>
<xsd:key name = "EmpKey">
  <xsd:selector xpath = "Dept/Emp" />
  <xsd:field xpath = "@eno" />
</xsd:key>
<xsd:keyref name = "DeptMgrFK" refer = "EmpKey">
  <xsd:selector xpath = "Dept" />
  <xsd:field xpath = "@mgr" />
</xsd:keyref>
</xsd:schema>
```





## XML Parsers

An ***XML parser*** processes the content and structure of an XML document and may use its schema (if one is present).

- Example: xmllint

XML parsers read in the XML document and determine if the document is well-formed and valid (if a schema is provided).

Once a document is parsed, programs may manipulate the document using one of two common interfaces: DOM and SAX.

- Note that you can write and parse XML documents without using a parser as the document itself is just a text file.
  - ⇨ DOM is a tree-based API that parses the entire document into an in-memory tree and provides methods for traversing the tree.
  - ⇨ SAX is an event-based serial access method. As a parser reads an XML file, it generates events when a new element is seen, closed, etc.





## XSL and XSLT

***XSL*** (eXtenstible Stylesheet Language) is a W3C recommendation that defines how XML data is displayed.

- It is similar but more powerful than Cascading Stylesheet Specification (CSS) used with HTML.

***XSLT*** (eXtenstible Stylesheet Language for Transformations) is a subset of XSL that provides a method for transforming XML (or other text documents) into other documents (XML, HTML).

# XSLT Overview

***XSLT*** works by defining transformation rules (templates) that match regions of the document specified by XPath expressions and then produce output for each matched region.

- This template matching is often done recursively.

Key syntax:

- Anything in { } is evaluated as an XPath expression.
- Template match (and recursively apply template):

```
<xsl:template match="Dept">
      <xsl:apply-templates select="Emp"/>
</xsl:template>
```

- Extract value:

```
<xsl:value-of select="name"/>
```





# XSLT Example

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:output method="xml" indent="yes"/>
  <xsl:template match="root">
      <Employees>
            <xsl:apply-templates select="Dept"/>
      </Employees>
  </xsl:template>
  <xsl:template match="Dept">
      <xsl:apply-templates select="Emp"/>
  </xsl:template>
  <xsl:template match="Emp">
      <Emp number="{@eno}">
            <xsl:value-of select="name"/>
      </Emp>
  </xsl:template>
</xsl:stylesheet>
```

XML output (indent)

action when match root element

match Dept under root

How to output attribute in HTML or XML output.

Outputs value of name tag as text in new Emp tag.





# XSLT Example (2)

```
<root>
  <Dept dno = "D1">
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <budget>350000</budget>
  </Dept>
</root>
```

**Output:**

```
<Employees>
   <Emp number="E7">R. Davis</Emp>
   <Emp number="E8">J. Jones</Emp>
   <Emp number="E6">L. Chu</Emp>
   <Emp number="E3">A. Lee</Emp>
</Employees>
```





# Conclusion

***Extensible Markup Language*** (***XML***) is a markup language that allows for the description of data semantics.

An XML document does not need a schema to be *well-formed*. An XML document is *valid* if it conforms to its DTD schema. XML Schemas can define schemas with more precision.

XML may be parsed (DOM/SAX), queried (XPath/XQuery) and displayed/transformed (XSL,XSLT).





# Objectives

- List some advantages of XML.
- Given an XML document, determine if it is well-formed.
- Given an XML document and a DTD, determine if it is valid.
- Explain the difference between #PCDATA and CDATA.
- Know the symbols (?,*,+) for cardinality constraints in DTDs.
- Compare and contrast ID/IDREFs in DTDs with keys and foreign keys in the relational model.
- List some advantages that XML Schema has over DTDs.
- Compare/contrast the two XML parser APIs: DOM and SAX.
- Explain why and when namespaces are used.
- Explain what XSL and XSLT are used for.
- Be able to write a XSLT document to transform an XML file.

# COSC 304
# Introduction to Database Systems

## XML Querying

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Querying XML

We will look at two standard query languages: XPath and XQuery.

XPath allows you to specify path expressions to navigate the tree-structured XML document.

XQuery is a full query language that uses XPath for path expressions.





## Example DTD

```
<!DOCTYPE Depts [
  <!ELEMENT Depts (Dept+)>
  <!ELEMENT Dept (name, Emp*, budget?)>
       <!ATTLIST Dept dno ID #REQUIRED>
       <!ATTLIST Dept mgr IDREF #IMPLIED>
  <!ELEMENT budget (#PCDATA)>
  <!ELEMENT Emp (name)>
       <!ATTLIST Emp eno ID #REQUIRED>
  <!ELEMENT name (#PCDATA)>
]>
```





## Example XML Document

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```





## Path Descriptions in XPath

***XPath*** provides the ability to navigate through a document using path descriptors.

***Path descriptors*** are sequences of tags separated by slashes /.

- If the descriptor begins with `/`, then the path starts at the root.
- If the descriptor begins with `//`, the path can start anywhere.
- You may also start the path by giving the document name such as `doc(depts.xml)/`.

A path descriptor denotes a sequence of nodes. These nodes may themselves contain other nodes (elements).

Examples:

- `/Depts/Dept/name`
- `//Dept/name`
- `doc("depts.xml")/Depts/Dept/Emp/name`

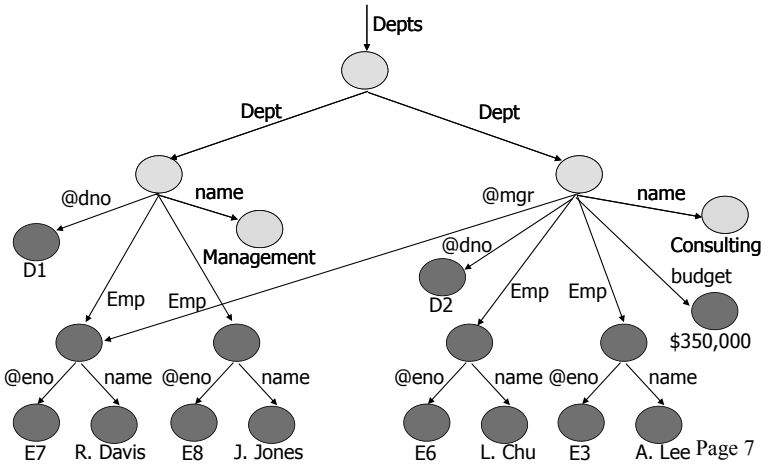




## Path: /Depts/Dept/name

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```

## Path: /Depts/Dept/name (tree view)








## Path: //Dept/name

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
      <name>Management</name>
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <name>Consulting</name>
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <budget>350000</budget>
  </Dept>
</Depts>
```

Path query returns same answer as previous one.





## Path: //name

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
      <name>Management</name>
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <name>Consulting</name>
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <budget>350000</budget>
  </Dept>
</Depts>
```

Matches any name tag starting from anywhere in the document.





## Path: /Depts/Dept

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
      <name>Management</name>
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <name>Consulting</name>
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <budget>350000</budget>
  </Dept>
</Depts>
```





## Wild Card Operator

The "*" wild card operator can be used to denote any ***single*** tag.

Examples:

- /*/*/name - Match any name that is nested 3 levels deep
- //* - Match anything





## Path: /*/*/name

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
      <name>Management</name>
      <Emp eno="E7"><name>R. Davis</name></Emp>
      <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
      <name>Consulting</name>
      <Emp eno="E6"><name>L. Chu</name></Emp>
      <Emp eno="E3"><name>A. Lee</name></Emp>
      <budget>350000</budget>
  </Dept>
</Depts>
```

Same as /Depts/Dept/name

## Question: What is /*/*/* ?

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```





## Attributes

Attributes are referenced by putting a "@" in front of their name.

Attributes of a tag may appear in paths as if they were nested within that tag.

Examples:

- `/Depts/Dept/@dno` - `dno` attribute of `Dept` element
- `//Emp/@eno` - `eno` attribute of `Emp` element





## Path: /Depts/Dept/@dno

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```





## Question: What is /*/*/@eno ?

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```





## Predicate Expressions

The set of objects returned can be filtered by putting selection conditions on the path.

A ***predicate expression*** may be specified inside square brackets **[**..**]** following a tag. Only paths that have that tag and also satisfy the condition are included in the result of a path expression.

Examples:

- `/Depts/Dept/name[.="Management"]`
- `//Depts/Dept[budget>250000]`
- `//Emp[@eno="E5"]`





## //Depts/Dept/budget[.>250000]

```
<?xml version = "1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE Depts SYSTEM "dept.dtd">
<Depts>
  <Dept dno = "D1">
       <name>Management</name>
       <Emp eno="E7"><name>R. Davis</name></Emp>
       <Emp eno="E8"><name>J. Jones</name></Emp>
  </Dept>
  <Dept dno = "D2" mgr = "E7">
       <name>Consulting</name>
       <Emp eno="E6"><name>L. Chu</name></Emp>
       <Emp eno="E3"><name>A. Lee</name></Emp>
       <budget>350000</budget>
  </Dept>
</Depts>
```

Note no budget element in first Dept so does not match path.

## Axes

Path expressions allow us to start at the root and execute a sequence of steps to find a set of nodes at each step. So far, we were always starting at a context node, and traversing edges to children nodes.

However, XPath defines several different axes that allow us to go from the current node to other nodes. An ***axis*** specifies the tree relationship between the nodes selected by the location step and the current node.

There are multiple different axes such as `parent::`, `child::`, `ancestor::`, and `descendant::` among others. All these define a set of nodes with the given relationship with the current node.





## XPath and Axes

Thus, evaluating an XPath expression amounts to starting with current nodes (called ***context nodes***) and then moving through the node hierarchy in a particular direction called an axis.

XPath evaluation description:

- When evaluating a path expression, the nodes selected in each step become the context nodes for the following step.
- If the input to a step is several context nodes, each is evaluated in turn. Evaluating a step involves:
  - ⇨Enumerating outgoing edges with matching labels AND
  - ⇨Only keeping destination nodes if they satisfy any predicate expression
- The result is output in the order of evaluation.





## Axes and Abbreviations

An axis to traverse is specified by putting the axis name before the tag name to be matched such as `child::Dept`.

Since this often results in long queries, some common axes have abbreviations:

- The default axis is `child::` which contains all children of a context node. Since it is the default, the child axis does not have to be explicitly specified.
  - ⇨Thus, `/Depts/Dept` is shorthand for `/Depts/child::Dept`
- **@** is a shorthand for the `attribute::` axis.
  - ⇨`/Depts/Dept/@dno` is short for `/Depts/Dept/attribute::dno`
- **..** is short for the `parent::` axis.
- **.** is short for the `self::` axis (current node).
- **//** is short for `descendant-or-self::` axis
  - ⇨// matches any node or any of its descendants





## Summary of XPath Constructs

| Symbol | Usage |
|---|---|
| / | Root element or separator between path steps |
| * | Match any single element name |
| @X | Match attribute X of current element |
| // | Match any descendant (or self) of current element |
| [C] | Evaluate condition on current element |
| [N] | Picks the *N*th matching element (indexed from 1) |
| parent:: | Matches parent element |
| descendant:: | Matches descendant elements |
| self:: | Matches the current element |
| ancestor:: | Matches ancestor elements |
| child:: | Matches children elements |
| node() | Matches any node (regardless of label) |





## DTD for Questions

```
<!DOCTYPE Bookstore [
  <!ELEMENT Bookstore (Book | Magazine)*>
  <!ELEMENT Book (Title, Authors, Remark?)>
  <!ATTLIST Book ISBN CDATA #REQUIRED>
  <!ATTLIST Book Price CDATA #REQUIRED>
  <!ATTLIST Book Edition CDATA #IMPLIED>
  <!ELEMENT Magazine (Title)>
  <!ATTLIST Magazine Month CDATA #REQUIRED>
  <!ATTLIST Year CDATA #REQUIRED>
  <!ELEMENT Title (#PCDATA)>
  <!ELEMENT Authors (Author+)>
  <!ELEMENT Remark (#PCDATA)>
  <!ELEMENT Author (First_Name, Last_Name)>
  <!ELEMENT First_Name (#PCDATA)>
  <!ELEMENT Last_Name (#PCDATA)>
]
```





## Example XML Document for Questions

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE Bookstore SYSTEM "bookstore.dtd">
<Bookstore>
   <Book ISBN="ISBN-0-201-70857-4" Price="65" Edition="3rd">
   <Title>Database Systems</Title>
   <Authors>
      <Author> <First_Name>Thomas</First_Name> <Last_Name>Connolly</Last_Name> </Author>
      <Author> <First_Name>Carolyn</First_Name> <Last_Name>Begg</Last_Name></Author>
    </Authors>
   </Book>
   <Book ISBN="ISBN-0-13-031995-3" Price="75">
   <Title>Database Systems: The Complete Book</Title>
   <Authors>
      <Author> <First_Name>Hector</First_Name><Last_Name>Garcia-Molina</Last_Name></Author>
      <Author> <First_Name>Jeffrey</First_Name> <Last_Name>Ullman</Last_Name> </Author>
     <Author> <First_Name>Jennifer</First_Name> <Last_Name>Widom</Last_Name> </Author>
   </Authors>
   <Remark> Amazon.com says: Buy these books together for a great deal!</Remark>
   </Book>
</Bookstore>
```

## XPath Questions

What are the elements selected by these XPath queries:

- `/Bookstore/*/Title`
- `//First_Name[.="Thomas"]`
- `//Last_Name[.="Ullman"]/parent::node()/parent::node()/parent::node()[@Price < 60]`

Write XPath queries to retrieve:

- all book titles
- all books < $70
- all last names anywhere
- all books containing a remark
- all book titles where the book < $80 and Ullman is an author
- retrieve the second book





## XQuery

***XQuery*** allows querying XML documents, using path expressions from XPath to describe important sets.

*FLWOR expressions* ("for-let-where-order by-return") are similar to SQL and consist of:

- One or more `for` and/or `let` clauses (bind variables)
  - ⇨FOR clause iterates through bound variables, while LET does not.
- An optional `where` clause (filters bound tuples)
  - ⇨Evaluated for each set of bound variables (tuple)
- An optional `order by` clause
- A `return` clause (generates output)
  - ⇨Executed once for each set of bound variables (tuple)

Variables begin with a dollar sign "$".





## XQuery *for* Clause Example

for loop variable

XPath expression to retrieve sequence of nodes to iterate over

```
for $en in /Depts/Dept/Emp/name
return <EmpName>{$en}</EmpName>
```

Brackets used to denote not regular text

```
Result:
<EmpName><name>R. Davis</name></EmpName>
<EmpName><name>J. Jones</name></EmpName>
<EmpName><name>L. Chu</name></EmpName>
<EmpName><name>A. Lee</name></EmpName>
```





## XQuery *let* Clause Example

```
let $en := /Depts/Dept/Emp/name
return <EmpName>{$en}</EmpName>
```

```
Result:
<EmpName>
  <name>R. Davis</name>
  <name>J. Jones</name>
  <name>L. Chu</name>
  <name>A. Lee</name>
</EmpName>
```





## XQuery WHERE Clause Example

```
for $e in /Depts/Dept/Emp
where $e/name > "I"
return <EmpName>{data($e/name)}</EmpName>
```

Returns data between tags instead of open/close tags and data together.

```
Result:
<EmpName>R. Davis</EmpName>
<EmpName>J. Jones</EmpName>
<EmpName>L. Chu</EmpName>
```





## XQuery FOR/LET Clause Example

Return all departments with at least 2 employees.

```
<result>
{
for $d in /Depts/Dept
where count($d/Emp) >= 2
return <DeptName>{data($d/name)}</DeptName>
}
</result>
```

```
Result:
<result>
<DeptName>Management</DeptName>
<DeptName>Consulting</DeptName>
</result>
```

## XQuery and IDREFs

In XQuery, it is possible to perform joins by using multiple XPath expressions.

A common case is to join an IDREF attribute with an ID attribute.





## XQuery IDREF Example

Print the manager name for each department.

```
for $d in /Depts/Dept,
      $e in /Depts/Dept/Emp[@eno = $d/@mgr]
return
<DeptMgr>
  <DeptName>{data($d/name)}</DeptName>
  <MgrName>{data($e/name)}</MgrName>
</DeptMgr>

Result:
<DeptMgr>
  <DeptName>Consulting</DeptName>
  <MgrName>R. Davis</MgrName>
</DeptMgr>
```





## XQuery Questions

Write XQuery queries to retrieve:

- 1) Return the book ISBN and price for each book.
- 2) Return only those books that have more than 2 authors.
- 3) Return average price of all books.
- 4) All titles of books costing < 80 where "Ullman" is an author.





## Conclusion

***XPath*** is a language for specifying paths through XML documents. An XPath expression enumerates a sequence.

***XQuery*** is a full query language based on XPath. The basis of XQueries is the FLWR expression.

- XPath is used to enumerate sequences of nodes.
- FOR is used to iterate through nodes, LET to store the entire sequence.
- WHERE is used to filter bindings and the RESULT clause specifies the output result.

XPath and XQuery are standards defined by the W3C.





## Objectives

- Given an XML document and query description, write an XPath query to retrieve the appropriate node sequence to answer the query.
- Given an XML document and an XPath expression, list the result of evaluating the expression.
- Be able to write simple XQuery queries given English text descriptions.

# COSC 304
# Introduction to Database Systems

## Data Warehousing

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## OLAP and OLTP

Most databases are designed to handle many small queries and updates. These databases are referred to as ***online transaction processing*** (***OLTP***) systems.

***Online analytical processing*** (***OLAP***) systems are designed for decision support applications where large amounts of data is analyzed often with *ad hoc* queries.

Comparison of OLTP versus OLAP:

| OLTP | OLAP |
|---|---|
| For transaction data | For historical data |
| Data is dynamic | Data is static |
| Common transactions | Ad hoc and varying queries |
| Transaction driven | Analysis driven |
| Large number of users | Smaller number of users |





## Data Warehousing

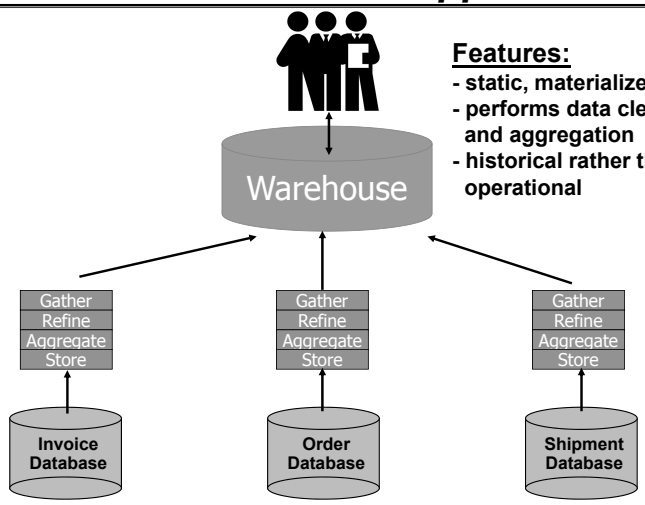
It is not practical to have long, analytical queries running on operational systems. One solution is to create a separate database that contains a copy of the operational data that is summarized and organized in an efficient manner.

A ***data warehouse*** is a historical database that summarizes, integrates, and organizes data from one or more operational databases in a format that is more efficient for analytical queries.

A ***data mart*** is a subset of a data warehouse that supports the business requirements of one department.



## Data Warehouse Approach




**Features:**
- **static, materialized view**
- **performs data cleansing and aggregation**
- **historical rather than operational**



## Building a Data Warehouse

The general steps to building a data warehouse are:

- 1) Identify the data available in operational systems that is necessary for analytical queries.
- 2) Build a model for the data warehouse, typically as a star schema.
- 3) Write extraction and cleansing routines (often using design tools) for exporting data from operational systems and importing it into the data warehouse.
  - ⇨ Data warehouses are often refreshed nightly.
- 4) Develop pre-made analysis reports and queries on the data warehouse and provide access to ad hoc query facility.
- 5) Formalize procedures for maintenance of the data warehouse and associated metadata including backup, refresh procedure, user access, and how changes to operational systems will be supported.





## Data Warehouse MetaData

***Metadata*** is data about data, or schema information.

In a data warehouse, metadata not only includes the type and name of fields, but also the source of the data, how and when it was extracted, and what transformations were applied to it before it was loaded into the warehouse.

Data warehouse metadata provides the history of every item in the warehouse. Maintaining accurate metadata is a critical documentation challenge.

# Data Warehouse Tools

There are tools available for supporting the construction of data warehouses. Using these tools is critical to the successful and timely deployment of a data warehouse.

The types of tools include:

- Extraction and Cleansing Tools
  - ⇨ Automate common tasks of extracting data from a data source, applying transformations, and then loading data into another data source.
- Report Design Tools
  - ⇨ For the construction of common reports.
- Analytical (OLAP) User Tools
  - ⇨ For supporting ad hoc querying to the data warehouse.
- Administration and Management Tools
  - ⇨ For backup and monitoring data warehouse performance.





# Star Schemas

Data warehouses are designed using a technique called ***dimensionality modeling*** where:

- A main table exists called a ***fact table*** with a composite primary key and foreign keys to one or more ***dimension tables***.
- A fact table consists of:
  - ⇨ *Dimension attributes*: key attributes of a dimension table.
  - ⇨ *Dependent attributes* : attributes determined by the dimension attributes

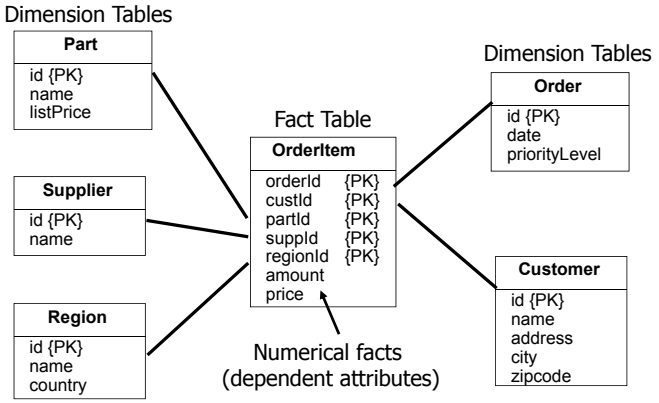
When drawn in a diagram, the schema looks like a star because the fact table is in the middle with multiple dimension tables linked to it. Hence, schemas for data warehouses are often called ***star schemas***.

The fact table is the largest table in the data warehouse and grows the fastest. All tables are treated as read-only.





# Example Star Schema








# OLAP Queries

A typical OLAP query on a star schema involves:

- a star join connecting the dimensions to the fact table
- selection of the data using the dimensions
- group by one or more dimensions
- aggregate the numeric facts/values

Example: For each customer in the city "New York", find the total number of products ordered from supplier "ABC":

```
SELECT   C.name, SUM(amount)
FROM     OrderItem OI, Customer C, Supplier S
WHERE    OI.custId = C.id and OI.suppId = S.id
         and S.name = 'ABC' and C.city = 'New York'
GROUP BY C.id, C.name;
```





# Other Design Issues

For efficiency and simplicity, natural keys such as a SSN or student # are replaced with autoincrement (integer) fields.

- This is done to save space and reduce the number of changes necessary if a key field happens to change. Such keys are called ***surrogate keys***.





# Data Warehouse Design Question

We have been using the following example WorksOn database:

emp (eno, ename, bdate, title, salary, supereno, dno)
proj (pno, pname, budget, dno)
dept (dno, dname, mgreno)
workson (eno, pno, resp, hours)

Assume the database stores the number of hours worked on a weekly basis. Each week the tuples in the WorksOn relation are deleted. Weeks are numbered by the company as 1,2,..,52.

Design a star schema for this data.

## Efficient Processing of OLAP Queries

There are some techniques that can speed the processing of OLAP queries.

***Materialized views*** are views that are physically stored in the database. Materialized views speed up OLAP queries by pre-computing large joins.

***Bitmap indexes*** allow for efficient lookup of data values. In bitmap indexing, a bit vector is created for each possible value of the attribute being indexed. The length of the bit vector is the number of tuples in the indexed table. The *j*-th bit of the vector is 1 if tuple *j* contains that value.

- Obviously, this is most effective when the domain of the attribute is small as a bit vector is needed for each value.





## Materialized View Example

Consider the query, for each customer in the city "New York", find the total number of products ordered from supplier "ABC":

```
SELECT   C.name, SUM(amount)
FROM     OrderItem OI, Customer C, Supplier S
WHERE    OI.custId = C.id and OI.suppId = S.id
         and S.name = 'ABC' and C.city = 'New York'
GROUP BY C.id, C.name;
```

Useful materialized view:

```
CREATE VIEW v1(custId, custName, suppName, amount) AS
SELECT   C.id, C.name, S.name, SUM(amount)
FROM     OrderItem OI, Customer C, Supplier S
WHERE    OI.custId = C.id and OI.suppId = S.id
GROUP BY C.id, C.name, S.id, S.name;
```





## Materialized View Example (2)

This view can be used to simplify the query to:

```
SELECT   C.name, SUM(v1.amount)
FROM     v1, Customer C
WHERE    v1.custId = C.id and v1.suppName = 'ABC'
         and C.city = 'New York'
GROUP BY C.id, C.name;
```

A view may significantly increase performance as the large join and aggregation involving the fact table is pre-computed.

Materialized views are especially effective in data warehouses because they are easy to maintain as the data is rarely changed except during loading.

- It is an active research area to determine what is the best set of views to materialize given a set of queries to answer.





## ROLAP versus MOLAP

There are two approaches to building a data warehouse:

- ***ROLAP*** - relational OLAP that uses a standard relational DBMS and star schema to store warehouse data.
- ***MOLAP*** - multidimensional OLAP that uses a specialized DBMS with a data cube model.

The advantage of ROLAP is that it can use existing DBMS technology. However, MOLAP systems can be more readily tuned for dimensional data.





## MOLAP and Data Cubes

The data structure in MOLAP is a ***data cube*** or ***hypercube***. A data cube can be thought of as a *N*-dimensional spreadsheet.

The keys of the dimension tables are the dimensions of the cube. Dependent attributes appear at the points of the cube. Thus, a tuple consists of the:

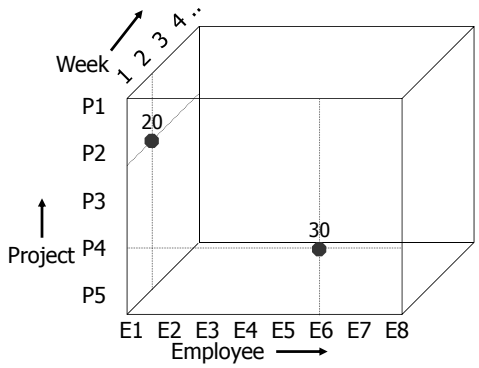
- key attributes - defines the position of the point in the cube
- dependent attributes - one or more values at that position in the cube

The data cube also includes aggregation along the margins of the cube. These ***marginals*** may include aggregations over one or more dimensions.



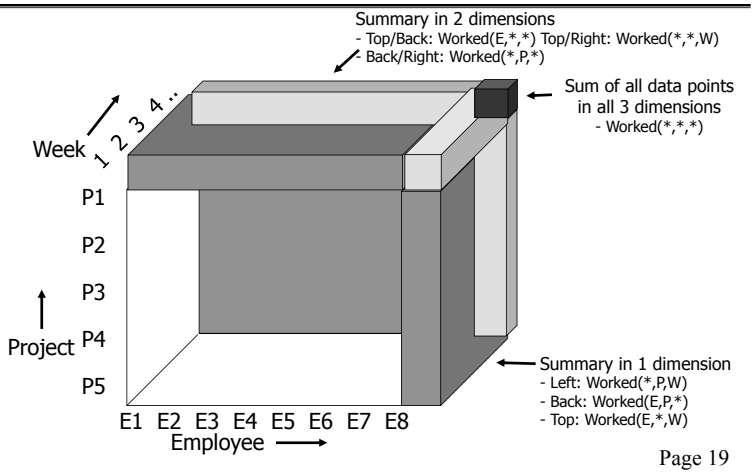


## Data Cube Example




Fact #1: Worked (E1, P2, 2, 20)
Fact #2: Worked (E6, P4, 1, 30)

## Data Cube Example Marginals








## Cube Aggregations

Think of each cube dimension as having an additional value * which stands for all values. A point with one or more *'s aggregates over the dimensions with the *'s.

Examples:

- `Worked("E1",*,*)` = sum of all hours `E1` has ever worked
- `Worked("E2","P3",*)` = sum of hours `E2` has worked on `P3`
- `Worked(*,*,*)` = sum of hours worked by all employees on all projects

Question: How would you answer these queries using the marginals in the data cube?





## Cube Operations

***Slice*** performs a select on one or more dimensions.

- Slice corresponds to adding a `WHERE` clause in SQL.

***Dice*** is used to partition or group on one or more dimensions

- Dice involves dividing a cube into smaller sub-cubes.
- Dice corresponds to adding a `GROUP BY` clause in SQL.





## Cube Navigation

Data analysts often "navigate" a data cube by summarizing data (roll-up) to get more general information or drilling-down to get more specific information.

***Drill-down*** is the process of dividing an aggregated value into its component parts.

- Example: `Worked("E1",*,*)` shows that `E1` has not worked too many hours. Can drill-down by project (2nd `*`) to show how many hours `E1` has worked per project.

***Roll-up*** is the process of aggregating along a particular dimension (opposite of drill-down).

- Example: `Worked("E1","P1",1)` shows `E1` has worked 20 hours in week1. Can roll-up on week: `Worked("E1","P1",*)` to get total hours `E1` has worked on `P1` over all weeks.





## What is Integration?

Two levels of integration:

- ***Schema integration*** is the process of combining local schemas into a global, integrated view by resolving conflicts present between the schemas.
- ***Data integration*** is the process of combining data at the entity-level. It requires resolving representational conflicts and determining equivalent keys.

Integration handles the different mechanisms for storing data (structural conflicts), for referencing data (naming conflicts), and for attributing meaning to the data (semantic conflicts).





## Integration Problems

Integration problems include:

- Different data models and conflicts within a model
- Incompatible concept representations
- Different user or view perspectives
- Naming conflicts (homonym, synonym)

## Integration using Mediators

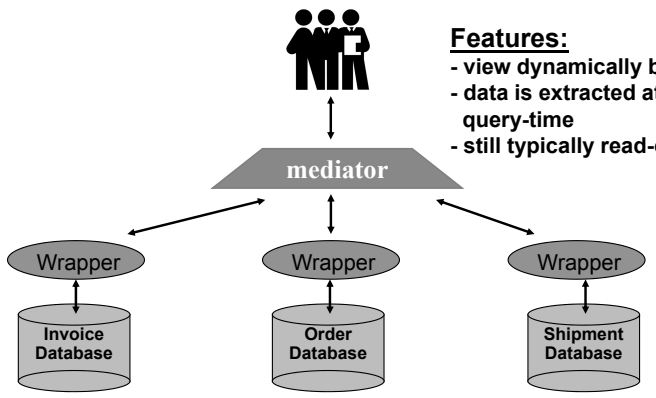
Unlike integration using a data warehouse, integration architectures that use wrappers and mediators provide online access to operational systems.

***Wrappers*** are software that converts global level queries into queries that the local database can handle. A ***mediator*** is global-level software that receives global queries and divides them into subqueries for execution by wrappers.

Unlike data warehouses, these systems are not suitable for decision-support queries because the data must be dynamically extracted from operational systems. They are useful for integrating operational systems without creating a single, unified database.



## Query-Driven Dynamic Approach




**Features:**
- **view dynamically built**
- **data is extracted at query-time**
- **still typically read-only**



## Integration Challenges

Database integration is an active area of research. Common problems include:

- 1) **Schema matching and merging** - How can we create a single, global schema for users to query? Can this be done automatically?
- 2) **Global Query Optimization** - How do we optimize the execution of queries over independent data sources?
- 3) **Global Transactions and Updates** - Is it possible to efficiently support transactions over autonomous databases?
- 4) **Global Query Languages** - Is SQL a suitable query language when the user does not understand the entire schema being queried?
- 5) **Peer-to-Peer** - What integration technologies are suitable for massive scale integrations over a grid or in dynamic peer-to-peer systems?





## Conclusion

A ***data warehouse*** is a historical database that summarizes, integrates, and organizes data from one or more operational databases in a format that is more efficient for analytical queries.

- OLAP differs from OLTP as it consists of longer, ad hoc queries that access more of the data in the database.

Building a data warehouse involves identifying operational sources, extracting and cleansing data, and loading the data into the warehouse.

Star schemas are used when a data warehouse is modeled in a relational DBMS. MOLAP stores the data in a cube and has operators such as slice, dice, roll-up, and drill-down.





## Objectives

- List some differences between OLTP and OLAP.
- Define data warehouse and describe some key features.
- Describe the steps in building a data warehouse.
- Explain how metadata is used in a data warehouse.
- Given a star schema, identify the fact table, dimension table, dimension attributes, and dependent attributes.
- List two ways the relational DBs speed-up processing OLAP queries (materialized view and bitmap indexes).
- Explain why materialized views are beneficial for warehouses.
- Compare the advantages of ROLAP versus MOLAP.
- Define the 2 cube operators: slice and dice.
- Explain cube navigation using drill-down and roll-up.
- Compare integration using mediators versus data warehousing.

# COSC 304
# Introduction to Database Systems

# Course Summary and COSC 404

**Dr. Ramon Lawrence**
**University of British Columbia Okanagan**
**ramon.lawrence@ubc.ca**



## Course Summary

The COSC 304 course goal was to:

Become an expert database user with the ability to query existing databases using SQL, design new databases using UML, and write programs that use databases.

High-demand skills acquired:

- Querying: SQL, relational algebra
- Database design: ER, UML
- Programming: Java, JDBC, PHP, HTML, JSON, XML, XPath
- Database skills make you more marketable and allow you to construct more sophisticated systems.
- All these skills practiced during the labs and project.





## Survey Question: Lecture Value

***Question:*** On a scale of 1 to 5 with 5 being the highest, how valuable/useful was the lecture time?

**A)** 1
**B)** 2
**C)** 3
**D)** 4
**E)** 5





## Survey Question: Lab Value

***Question:*** On a scale of 1 to 5 with 5 being the highest, how valuable/useful was the lab time and assignments?

**A)** 1
**B)** 2
**C)** 3
**D)** 4
**E)** 5





## Survey Question: Workload

***Question:*** On a scale of 1 to 5 with 1 being very low and 5 being very high, how was the overall workload compared to other courses and your expectations?

**A)** 1
**B)** 2
**C)** 3
**D)** 4
**E)** 5





## Survey Question: Clicker Value

***Question:*** On a scale of 1 to 5 with 5 being the highest, how valuable/useful were the clicker questions used in-class?

**A)** 1
**B)** 2
**C)** 3
**D)** 4
**E)** 5

## COSC 304 vs. COSC 404

COSC 304
Introduction to Database Systems

**Database Design and Programming**
- Data models - ER, relational, XML, JSON
- Query languages - SQL, relational algebra
- Design project
- Database skills and techniques as a user
- How to use a DBMS ; how to build a database

COSC 404
Database System Implementation

**Database System Implementation**
- Storage and index structures
- Transaction management, concurrency control
- Query processing, recovery and reliability
- How to build a DBMS
- Non-relational systems and architectures
- How to select a DBMS





## COSC 404 Course Goals

COSC 404 is about how a database works (the "**black box**").
- Inside is storage, indexing, query processing/optimization, transactions, concurrency, recovery, distribution, lots of stuff!

Goals:

1) Be a better, "expert" user of database systems.
2) Be able to use and compare different database systems.
3) Adapt the techniques when developing your own software.

You will gain ***lots*** of industrial experience using a variety of databases and became a better, more experienced developer.
- MySQL, PostgreSQL, Microsoft SQL Server, MongoDB, JUnit, VoltDB, Java, JDBC, javacc, JSON, Map-Reduce, SQL





# Thank you for a great course!

**Good luck on the exam!**