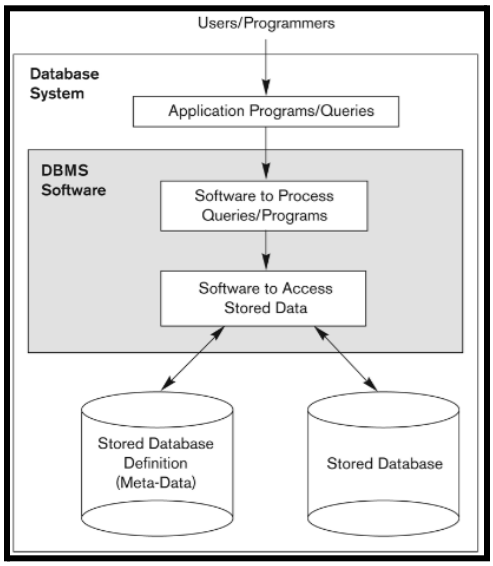
CS 4400

Module 1: Administration & Structure



- Databases are having a growing importance in an age where data is king
- DEFINITION: A **database** is a collection of related *data*, which are known facts that can be recorded and have implicit meaning
- DEFINITION: A **database management system** (**DBMS**) is a software package/system which facilitates the creation and maintenance of a computerized database
    - A *database system* is the DBMS with the data itself
- DEFINITION: A **mini-world** is some part of the real world about which data is stored in a database
    - Ex. Student grades and transcripts at a university
- Applications interact with a database by…
    - *Querying*: access different parts of data and formulate the result of a request
    - *Transacting*: Read some data and "update" certain values or generate new data and store that
- Here is an example of a database…
    - Suppose our *mini-world* is part of a UNIVERSITY environment. Some *entities* we may see include…
        - STUDENT
        - COURSE
        - DEPARTMENT
        - INSTRUCTOR
    - Some relationships we may see include…
        - COURSE have *prerequisite* COURSE
        - INSTRUCTION *teach* COURSE
        - COURSE *offered by* DEPARTMENT
        - STUDENT *major in* DEPARTMENT

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Module 2: MySQL Intro, Basic Database Terminology & Architecture

- DEFINITION: **Data models** are sets of concepts to describe…
    - the ***structure*** of a database,
    - the ***operations*** for manipulating these structures,
    - and certain ***constraints*** that the database should obey
- *Structure* is given by how we define the tables in our database
    - "Tables are the fundamental structures in relational database systems; tables hold data like a glass holds water"
    - A table is a collection of records and each record fits a certain structure and has a certain number of columns (attributes); we put data in those columns, and data must obey certain data type rules

course_name	course_number	credit_hours	department
Intro To Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Database	CS3380	3	CS
Discrete Mathematics	MATH2410	3	MATH

- *Operations* allow us to manipulate structures and give us ways to uniquely view our structures
    - See a cheat sheet for SQL here
- *Constraints* prevent us from corrupting or doing bad things to our data based on rules that we set
    - Constraints typically include…
        - *Elements* (and their *data types*)
        - Groups of *elements* (*entity, record, table*)
        - *Relationships* amongst groups
    - Constraints specify some restrictions on valid data and must be enforced at all times
- There are four different categories of data models…
    - *Conceptual* (high-level, semantic): Provide concepts that are close to the way many user perceive data
    - *Physical* (low-level, internal): Provide concepts that describe details of how data is stored in the computer, specified in an ad-hoc manner through DBMS design and administration manuals
    - *Implementation* (representational): Provide concepts that fall between the above two
    - *Self-Describing*: Combine the description of data with the data values
        - Ex. XML, key-value stores and some NOSQL systems

MySQL Commands

Sources

- W3Schools.com
- DataQuest.io

SQL CHEATSHEET

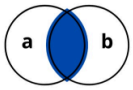
CONSIDER SUPPORTING ME

@AbzAaron

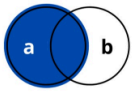
Commands / Clauses

SELECT	Select data from database
FROM	Specify table we're pulling from
WHERE	Filter query to match a condition
AS	Rename column or table with alias
JOIN	Combine rows from 2 or more tables
AND	Combine query conditions. All must be met
OR	Combine query conditions. One must be met
LIMIT	Limit rows returned. See also FETCH & TOP
IN	Specify multiple values when using WHERE
CASE	Return value on a specified condition
IS NULL	Return only rows with a NULL value
LIKE	Search for patterns in column
COMMIT	Write transaction to database
ROLLBACK	Undo a transaction block
ALTER TABLE	Add/Remove columns from table
UPDATE	Update table data
CREATE	Create TABLE, DATABASE, INDEX or VIEW
DELETE	Delete rows from table
INSERT	Add single row to table
DROP	Delete TABLE, DATABASE, or INDEX
GROUP BY	Group data into logical sets
ORDER BY	Set order of result. Use DESC to reverse order
HAVING	Same as WHERE but filters groups
COUNT	Count number of rows
SUM	Return sum of column
AVG	Return average of column
MIN	Return min value of column
MAX	Return max value of column

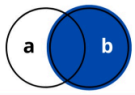
Joins



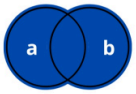
a INNER JOIN b



a LEFT JOIN b



a RIGHT JOIN b



a FULL OUTER JOIN b

Examples

Select all columns with filter applied

```
SELECT * FROM tbl
WHERE col > 5;
```

Select first 10 rows for two columns

```
SELECT col1, col2
FROM tbl LIMIT 10;
```

Select all columns with multiple filters

```
SELECT * FROM tbl
WHERE col1 > 5 OR col2 < 2;
```

Select all rows from col1 & col2 ordering by col1

```
SELECT col1, col2
FROM tbl ORDER BY 1;
```

Return count of rows in table

```
SELECT COUNT(*)
FROM tbl;
```

Return sum of col1

```
SELECT SUM(col1)
FROM tbl;
```

Return max value for col1

```
SELECT MAX(col1)
FROM tbl;
```

Compute summary stats by grouping col2

```
SELECT AVG(col1) FROM tbl
GROUP BY col2;
```

Combine data from 2 tables using left join

```
SELECT * FROM tbl1 AS t1 LEFT JOIN
tbl2 AS t2 ON t2.col1 = t1.col1;
```

Aggregate and filter result

```
SELECT col1,
       COUNT(*) AS total
FROM tbl
GROUP BY col1
HAVING COUNT(*) > 10;
```

Implementation of CASE statement

```
SELECT col1,
CASE
    WHEN col1 > 10 THEN 'more than 10'
    WHEN col1 < 10 THEN 'less than 10'
    ELSE '10'
END AS NewColumnName
FROM tbl;
```

Data Definition Language

CREATE

```
CREATE DATABASE MyDatabase;

CREATE TABLE MyTable (
    id int,
    name varchar(10));

CREATE INDEX IndexName
ON TableName(col1);
```

ALTER

```
ALTER TABLE MyTable
DROP COLUMN col5;

ALTER TABLE MyTable
ADD col5 int;
```

DROP

```
DROP DATABASE MyDatabase;
DROP TABLE MyTable;
```

Data Manipulation Language

UPDATE

```
UPDATE MyTable
SET col1 = 56
WHERE col2 = 'something';
```

INSERT

```
INSERT INTO MyTable (col1, col2)
VALUES ('value1', 'value2');
```

DELETE

```
DELETE FROM MyTable
WHERE col1 = 'something';
```

SELECT

```
SELECT col1, col2
FROM MyTable;
```

Order Of Execution

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY
7. LIMIT

Schemas v. States

- DEFINITION: **Schemas** are descriptions of a database, which includes descriptions of the database structure, data types, and the constraints of the database. Sort of like the *object class* in OOP.
    - DEFINITION: A **schema diagram** is an illustrative display of (most aspects of) a database schema
    - DEFINITION: A **schema construct** is a component of the schema or an object within the schema
        - E.g. STUDENT, COURSE

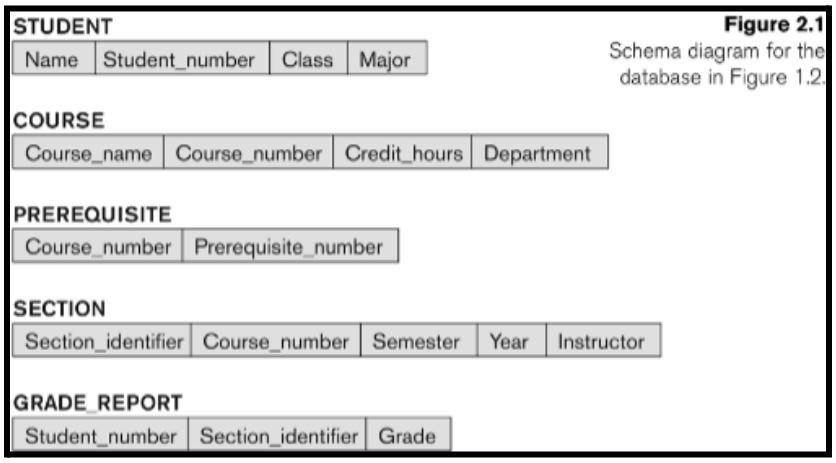


Figure 2.1
Schema diagram for the database in Figure 1.2.

- DEFINITION: The **state** of the database (database instance/occurrence/snapshot) is the actual data stored in a database at a *particular moment in time*, including the collection of all the data in the database
    - The *initial database state* is the database state when it is initially loaded into the system
    - A *valid state* is a state that satisfies the structure and constraints of the database

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

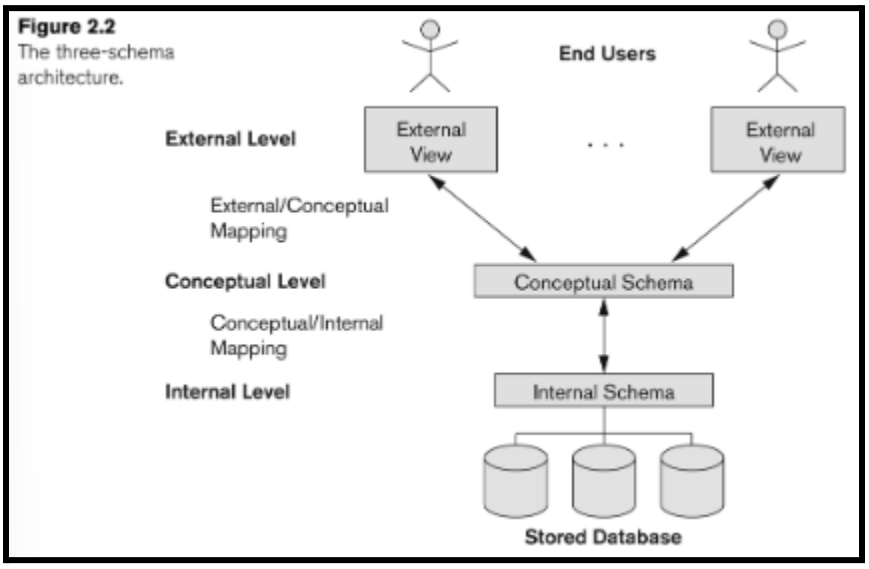
SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	04	King
92	CS1310	Fall	04	Anderson
102	CS3320	Spring	05	Knuth
112	MATH2410	Fall	05	Chang
119	CS1310	Fall	05	Anderson
135	CS3380	Fall	05	Stone

- Database schema changes very infrequently, database state changes all the time

Three-Schema Architecture

- DEFINITION: The **three-schema architecture** is a proposed architecture useful for explaining database system organization. Defines DBMS schemas at three levels…
    - *Internal Schema*: Describe physical storage structures and access paths (e.g. indexes)
        - Typically use a *physical* data model
    - *Conceptual Schema*: Describe the structure and constraints for the whole database for a community of users
        - Uses a *conceptual* or *implementation* data model
    - *External Schema*: Describe the various user views



Data Independence

- There are a couple of types of data independence…
    - *Logical*: The capacity to change the conceptual schema without changing the external schemas and their associated application programs
    - *Physical*: The capacity to change the internal schema without having to change the conceptual schema
- When a lower-level schema is changed, only the mappings between the schema and higher-levels need to be changed in a DBMS that fully supports data independence

DBMS Languages

- DEFINITION: A **data definition language** (**DDL**) use used to specify the conceptual schema of a database and often times are used to define internal and external schemas
- DEFINITION: A **data manipulation language** (**DML**) is used to specify database retrievals and updates. There are two types…
    - *High Level* (*Non-Procedural/Declarative*) *Language*: Are "set"-oriented and specify what data to retrieve rather than how to retrieve it (e.g. SQL relational

language)

- *Low Level* (*Procedural*) *Language*: Retrieve data one record-at-a-time; uses constructs such as loops to retrieve multiple records
    - Can be embedded in a general-purpose programming language like C, C++, or Java
    - Alternatively, stand-alone DML commands can be applied directly (called a *query language* [sound familiar?])

Module 3: Select, From, Where SQL Queries

- DEFINITION: **Table**, **row**, and **column** are used for relational model terms *relation*, *tuple*, and *attribute*

Data Types

- SQL, like many languages, supports a variety of data types
- Numeric Data Types
  - *Integers*: `INTEGER`, `INT`, and `SMALLINT`
  - *Floating-Points*: `FLOAT` or `REAL`, and `DOUBLE PRECISION`
- Character-String Data Types
  - *Fixed Length*: `CHAR(n), CHARACTER(n)`
  - *Varying Length*: `VARCHAR(n), CHAR VARYING(n), CHARACTER VARYING(n)`
- Bit-String Data Types
  - *Fixed Length*: `BIT(n)`
  - *Varying Length*: `BIT VARYING(n)`
- Boolean Data Type
  - Values of `TRUE` or `FALSE` or `NULL`
- DATE Data Type
  - Has ten positions with the component of `YEAR`, `MONTH`, and `DAY` in the form of YYYY-MM-DD
- Timestamp Data Type
  - Includes `DATE` and `TIME` fields, plus a minimum of six positions for decimal fractions of seconds with an optional `WITH TIME ZONE` qualifier
- INTERVAL data type
  - Specifies a relative value that can be used to increment/decrement an absolute value of a date, time, or timestamp
- NOTE: `DATE`, `TIME`, Timestamp, and `INTERVAL` data types can be cast to strings for comparison

Domains

- DEFINITION: A **domain** is a data type with optional constraints defined by the user
  - Ex. `CREATE DOMAIN SSN_TYPE AS CHAR(9);` creates the domain `SSN_TYPE`, which can now be used in place of a data type and constricts the data to be a character of length 9
  - Ex. `CREATE DOMAIN CPI_DATA AS REAL CHECK (value >= 0 AND value <= 10);` creates the domain `CPI_DATA`, which constricts the data to be a floating point in the range [0, 10]

- Makes it easier to change the data type for a domain that is used by numerous attributes
- Improves database readability

Constraints in SQL

- The relational model has 3 constraint types…
  - *Key Constraint*: A primary key value cannot be duplicated
  - *Entity Integrity Constraint*: A primary key value cannot be null
  - *Referential Integrity Constraints*: The "foreign key" must have a value that is already present as a primary key, or may be null
- There are other restrictions on attributes that can be used…
  - `DEFAULT <value>`, indicates the default value for an attribute
  - `NOT NULL`, indicates that `NULL` is not permitted
  - `CHECK`, indicates that the attribute is only valid if it passes a specified check
    - e.g. `Dnumber INT NOT NULL CHECK (Dnumber > 0 AND Dnumber < 21);`
  - `PRIMARY KEY`, specifies one or more attributes that make up the primary key of a relation
    - e.g. `Dnumber INT PRIMARY KEY;`
  - `UNIQUE`, specifies alternate (secondary) keys (called CANDIDATE keys in the relational model)
    - e.g. `Dname VARCHAR(15) UNIQUE;`
  - `FOREIGN KEY`
- Similar to how we could group limitations for a data type using domains, we can group constraints into one constraint using the keyword `CONSTRAINT`, which makes updating a constraint in the future much easier
- Additional constraints on individual tuples within a relation are also possible using `CHECK`
  - e.g. `CHECK (Dept_create_date <= Mgr_start_date);`

Querying in SQL

SELECT <attribute list>
FROM <table list>
WHERE <condition>;

where

- <attribute list> is a list of attribute names whose values are to be retrieved by the query.
- <table list> is a list of the relation names required to process the query.
- <condition> is a conditional (Boolean) expression that identifies the tuples to be retrieved by the query.

- We can still use the above query without the `WHERE` clause, as it simply indicates there's no condition on tuple selection
- Selecting multiple attributes from multiple relations results in all possible tuple combinations

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:   SELECT   Ssn
      FROM     EMPLOYEE;

Q10:  SELECT   Ssn, Dname
      FROM     EMPLOYEE, DEPARTMENT;
```

- Using an asterisk (*) in `SELECT` will retrieve all the attribute values of the selected tuples (in English, you can say "all")

```
Q1C:  SELECT   *
      FROM     EMPLOYEE
      WHERE    Dno=5;

Q1D:  SELECT   *
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dno=Dnumber;

Q10A: SELECT   *
      FROM     EMPLOYEE, DEPARTMENT;
```

Alleviating Ambiguity

- Suppose we have two attributes which have the same name in different relations (i.e. both EMPLOYEE and DEPARTMENT have Dnumber), and we want to select one of the attributes when querying from both relations
- We must qualify our attribute name with the relation name

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Aliases and Renaming

- We can declare alternative relation name to refer to the same relation twice in a query

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM    EMPLOYEE AS E, EMPLOYEE AS S
WHERE   E.Super_ssn=S.Ssn;
```

- Attribute names can also be renamed

```
EMPLOYEE AS E(Fn, Mi, Ln, Ssn, Bd,
 Addr, Sex, Sal, Sssn, Dno)
```

Tables as Sets in SQL

- Tables in SQL can be treated as sets. As such, we can use a couple of interesting operations that may be useful
- Operations
    - You can use the keyword `DISTINCT` or `ALL` in the `SELECT` clause to indicate that you want don't or do want to include duplicate tuples respectively

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:   SELECT    ALL Salary
       FROM      EMPLOYEE;

Q11A:  SELECT    DISTINCT Salary
       FROM      EMPLOYEE;
```

    - You can use UNION, EXCEPT (difference), INTERSECT from set theory

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
Q4A:  ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, DEPARTMENT, EMPLOYEE
        WHERE     Dnum=Dnumber AND Mgr_ssn=Ssn
                  AND Lname='Smith' )
        UNION
      ( SELECT    DISTINCT Pnumber
        FROM      PROJECT, WORKS_ON, EMPLOYEE
        WHERE     Pnumber=Pno AND Essn=Ssn
                  AND Lname='Smith' );
```

Substring Pattern Matching and Arithmetic Operations

- `LIKE` comparison operator
    - Used for string pattern matching
    - % = arbitrary number of zero or more characters
    - _ = single character
    - Examples
        - `WHERE Address LIKE '%Houston, TX%';`
        - `WHERE Ssn LIKE '__1__8901';`
- BETWEEN comparison operator
    - Ex. `WHERE(Salary BETWEEN 30000 AND 40000);`
- Standard arithmetic operations like addition (+), subtraction (-), multiplication (*), and division (/) may be included as part of `SELECT`

Query 13. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

```
SELECT  E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM    EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE   E.Ssn=W.Essn AND W.Pno=P.Pnumber AND
        P.Pname='ProductX';
```

Ordering of Query Results

- Use ORDER BY clause
  - DESC, descending order
  - ASC, ascending order
  - Typically at end of query

```
ORDER BY D.Dname DESC, E.Lname ASC,
  E.Fname ASC
```

- With this, we now have an updated SQL retrieval query structure (updated later)

```
SELECT      <attribute list>
FROM        <table list>
[ WHERE     <condition> ]
[ ORDER BY <attribute list> ];
```

Modifying a Database

- We have three commands to modify a database…
  - `INSERT`, which typically inserts a tuple (row) in a relation (table)
    - Attribute values should be listed in the same order as the attributes were specified in the CREATE TABLE command
    - Constraints on data types are observed automatically
    - When inserting, specify the relation name and a list of values for the tuple

```
U1:   INSERT INTO   EMPLOYEE
      VALUES        ( 'Richard', 'K', 'Marini', '653298653', '1962-12-30', '98
                      Oak Forest, Katy, TX', 'M', 37000, '653298653', 4 );
```

    - You can insert results from a query…

```
U3B:  INSERT INTO   WORKS_ON_INFO ( Emp_name, Proj_name,
                    Hours_per_week )
      SELECT        E.Lname, P.Pname, W.Hours
      FROM          PROJECT P, WORKS_ON W, EMPLOYEE E
      WHERE         P.Pnumber=W.Pno AND W.Essn=E.Ssn;
```

  - `UPDATE`, which may update a number of tuples (rows) in a relation (table) that satisfy the condition
    - WHERE clause selects the tuples to be modified
    - An additional SET clause specifies the attributes to be modified and their new values
    - Ex. Change the location and controlling department number of project number 10 to “Bellaire” and 5, respectively

```
U5:     UPDATE    PROJECT
        SET       PLOCATION = 'Bellaire',
                      DNUM = 5
        WHERE     PNUMBER=10
```

- `DELETE`, which may also update a number of tuples (rows) in a relation (table) that satisfy the condition
  - Includes a WHERE clause to select the tuples to be deleted
  - A missing WHERE clause specifies that *all tuples* the the relation are to be deleted; becomes an empty table

```
U4A:    DELETE FROM    EMPLOYEE
        WHERE          Lname='Brown';

U4B:    DELETE FROM    EMPLOYEE
        WHERE          Ssn='123456789';

U4C:    DELETE FROM    EMPLOYEE
        WHERE          Dno=5;

U4D:    DELETE FROM    EMPLOYEE;
```

Module 4: SQL Data Definitions, Data Types and Domain Constraints

```
DROP TABLE IF EXISTS employee;
CREATE TABLE employee (
  fname char(10) NOT NULL,
  lname char(20) NOT NULL,
  ssn decimal(9,0) NOT NULL,
  bdate date NOT NULL,
  address char(30) NOT NULL,
  sex char(1) NOT NULL,
  salary decimal(5,0) NOT NULL,
  superssn decimal(9,0) DEFAULT NULL,
  dno decimal(1,0) NOT NULL
);
```

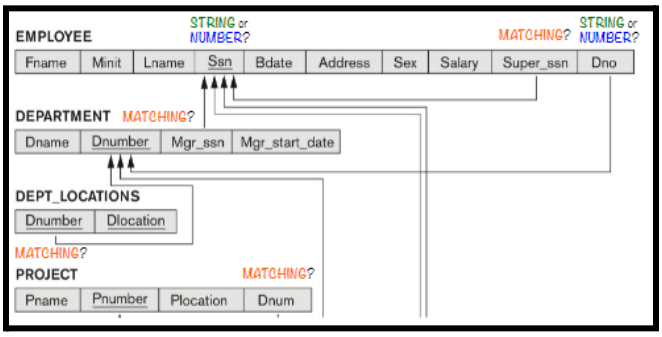
- This module is an extension of the previous module, Module 3
- Every column has a specific *data type* associated with it
- DEFINITION: A **data type** tells the database system what types of values that will be stored in a particular column of a particular table of the database
- Data types have 3 main data type categories…
    - *Numbers*
    - *Dates*
    - *Words*
        - NOTE: See previous module for more specifics on the exact data types SQL has
- When deciding what data type to use for a particular column, there are three basic principles to use…
    - *Correctness*
    - *Consistency*
    - *Efficiency*

Correctness

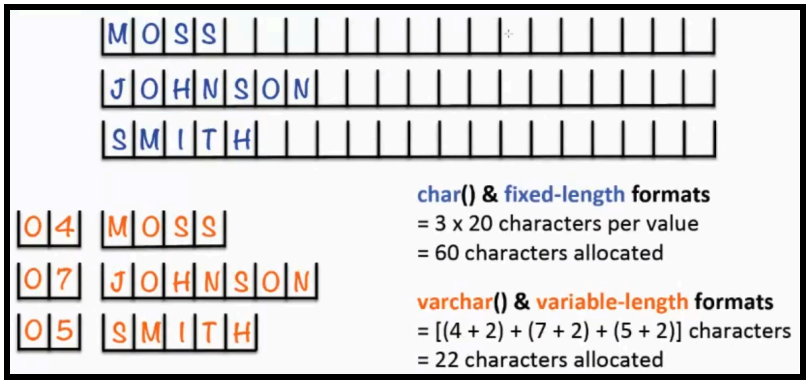
- Ensure that you select the correct type to store the required data
    - Ex. For a numeric value…
        - Is it a whole number, or does it have a decimal component?
        - What is the maximum size of the value?
        - How precise is the value?
        - How detailed is the decimal component?
        - Be wary of *automatic conversions*
- "Pick a data type that will not compromise the integrity of the data you're storing"

Consistency

- Attributes that will be compared should have the same data type when possible
- If attributes are connected across different tables, they should share the same data type



Efficiency



- There are different ways to store the same data
    - Ex. Look at our `CHAR` and `VARCHAR` data types, both store strings
- Choosing between them means we are interested in different performance characteristics
    - Ex. Speed vs. precision
- Here are some different metrics people can care about…
    - *Space Efficiency*: I want to be able to cram as much data as possible in as little space as possible
    - *Speed Efficiency*: I want to be able to access the data as fast as possible

Interesting Quirk about the `DATE` Data Type

```
select '2021-09-02';

select 2021-09-02;
```

- When we want to refer to a date in SQL, we have to be cognizant of the fact that SQL can handle both dates and basic arithmetic operations
- The top query returns the result `2021-09-02`, however the bottom one returns the result 2010
    - This is because the top query is the correct way of referring to a `DATE` data type, while the bottom query is an arithmetic expression

Schema and Catalog Concepts in SQL

- SQL schema is identified by a *schema name* and includes an *authorization identifier* and *descriptors* for each element
- Schema elements include…
    - *Tables*
    - *Constraints*
    - *Views*
    - *Domains*
    - etc.
- `CREATE SCHEMA` statement
    - `CREATE SCHEMA COMPANY AUTHORIZATION "Jsmith";`

The CREATE TABLE Command in SQL

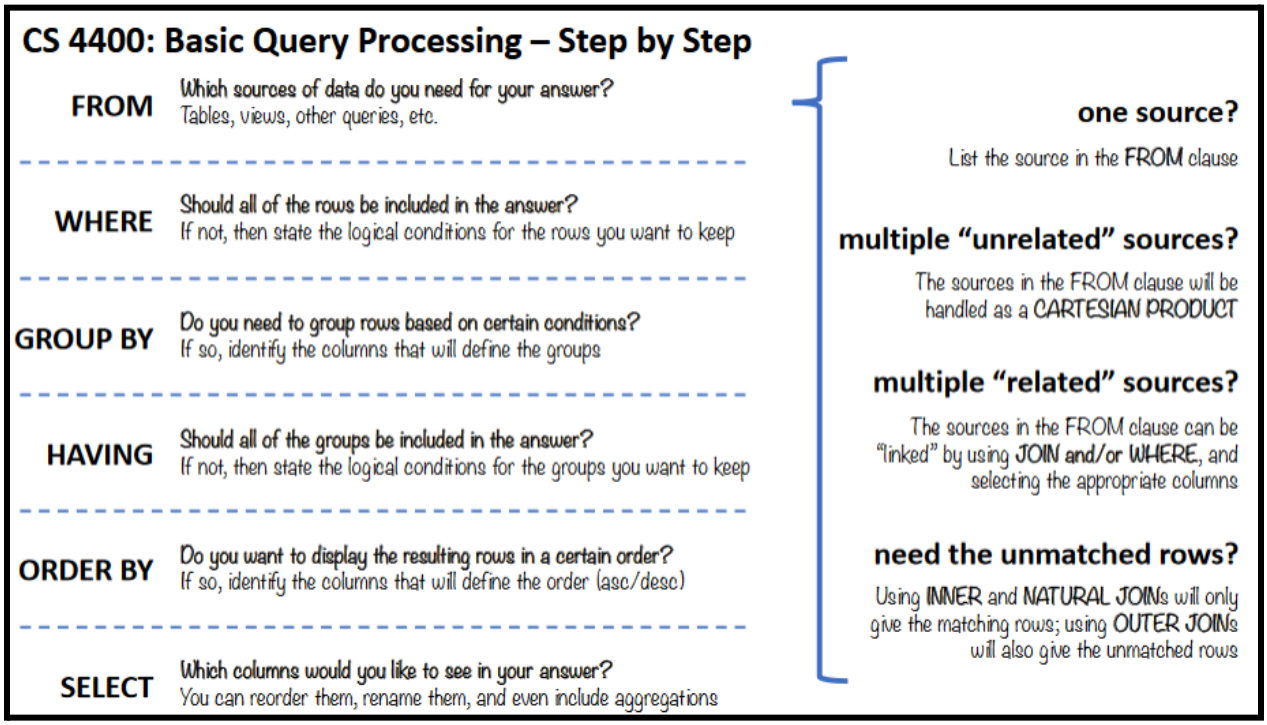
- Tables in SQL pertain to a relation
- When creating a table, provide the name and specify attributes, their types and initial constraints
- Tables can optionally specify the schema they are a part of…
    - `CREATE TABLE COMPANY.EMPLOYEE …`

```
CREATE TABLE EMPLOYEE
        ( Fname                 VARCHAR(15)                 NOT NULL,
          Minit                 CHAR,
          Lname                 VARCHAR(15)                 NOT NULL,
          Ssn                   CHAR(9)                     NOT NULL,
          Bdate                 DATE,
          Address               VARCHAR(30),
          Sex                   CHAR,
          Salary                DECIMAL(10,2),
          Super_ssn             CHAR(9),
          Dno                   INT                         NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname                 VARCHAR(15)                 NOT NULL,
          Dnumber               INT                         NOT NULL,
          Mgr_ssn               CHAR(9)                     NOT NULL,
          Mgr_start_date        DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber               INT                         NOT NULL,
          Dlocation             VARCHAR(15)                 NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
```

Tables v. Views

- Although we will go more into this later, for now just remember that…
    - *Tables* : Represent a relation; the relation and its tuples are actually created and stored as a filed by the DBMS
    - *Views* : Created through the `CREATE VIEW` statement and do *not* correspond to any physical file

Module 5: Aggregation & Group By in SQL



- The above are the "Big 6" things to account for when writing a query; note that not every section must be filled out when making a query
- Previously, we saw a structured way of formatting out database queries. The above figure expands that format to encompass some of the more rigorous, advanced concepts we can use in querying
    - NOTE: There isn't *really* a true format to writing a query, but this figure can help us think of walking through queries step-by-step, but it will also assist us in thinking of a concept we'll see in a moment
- Also previously, our order was SELECT-FROM-WHERE. Why did we change it to this? Because this is the way relational database systems process your queries
    - When you type SELECT-FROM-WHERE, the database doesn't start limiting columns due to your SELECT input, and how can it? It doesn't know where it's limiting!

NULL Logic

- The NULL value can have three different meanings depending on the situation…
    - Unknown value
    - Unavailable or withheld value
    - Not applicable attribute
- Each individual NULL value is considered to be different from every other NULL value
    - As such, the NULL = NULL comparison is often avoided

- SQL uses three-valued logic for booleans…
  - `TRUE`
  - `FALSE`
  - `UNKNOWN`

Table 7.1 Logical Connectives in Three-Valued Logic

(a)	**AND**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	**OR**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	**NOT**			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

- Although we should generally avoid comparing `NULL`'s, SQL has no problem affirming whether an attribute is or is not `NULL`…

`IS` or `IS NOT NULL`

Aggregate Functions in SQL

- SQL has a variety of built-in aggregate functions we can use
  - See under `GROUP BY` and `ORDER BY` in picture here

Renaming Results of Aggregation

- Using AS, we can rename the results of our aggregation
  - Ex. Suppose we do the following query…

```
SELECT  SUM (Salary), MAX (Salary), MIN (Salary), AVG
        (Salary)
FROM    EMPLOYEE;
```

  - This would result in a single row of computed values titled `SUM(Salary), MAX(Salary), MIN(Salary), and AVG(Salary)`
  - These row titles could get pretty if they had big computations inside of them. Using AS, we can rename them…

```
SELECT   SUM (Salary) AS Total_Sal, MAX (Salary) AS
         Highest_Sal, MIN (Salary) AS Lowest_Sal, AVG
         (Salary) AS Average_Sal
FROM     EMPLOYEE;
```

- When using these aggregate functions, NULL's are discarded
- There are also aggregate functions on booleans…
    - `SOME` (OR) : returns true if at least one element is TRUE
    - `ALL` (AND) : returns true if all elements are true

Grouping: The GROUP BY Clause and HAVING Clauses

- The `GROUP BY` clause specifies grouping attributes, can be used to partition relation into subsets
- The grouping attribute *must* appear in the SELECT clause

```
SELECT      Dno, COUNT (*), AVG (Salary)
FROM        EMPLOYEE
GROUP BY    Dno;
```

- If the grouping attribute has NULL as a possible value, then a separate group is created for the null value
- The HAVING clause provides a condition to select or reject an entire group

Query 26. For each project *on which more than two employees work*, retrieve the project number, the project name, and the number of employees who work on the project.

```
Q26:    SELECT      Pnumber, Pname, COUNT (*)
        FROM        PROJECT, WORKS_ON
        WHERE       Pnumber=Pno
        GROUP BY    Pnumber, Pname
        HAVING      COUNT (*) > 2;
```

- You can create subgroups of tuples before summarizing using the `HAVING` clause
    - In practice, HAVING and WHERE seem like they do the same thing. In essence they do; they are both clauses that act as filters and remove records or data that don't meet certain criteria
    - The difference lies in the level they filter at
        - WHERE filters at the *record level*, i.e. individual rows
        - HAVING filters at the *"group of records" level*, i.e. aggregations, which can be the results of a selection from things like COUNT, MAX, MIN, etc.

Module 6: Entity & Attribute Concepts

- The focus of this chapter is on conceptual database design and how to design the conceptual schema for a database application

Entity-Relationship (ER) Model Concepts

- DEFINITION: An **entity** is a basic concept for the ER model; they are specific things or objects in the mini-world that are represented in the database (like nouns)
    - Ex. The EMPLOYEE John Smith, the Research DEPARTMENT
    - NOTE: Entities with the same basic attributes are grouped or typed into an *entity type* (ex. EMPLOYEE or PROJECT)
- DEFINITION: **Attributes** are properties used to describe an entity (not necessarily adjectives, but more-so the details of an entity). There are three types of attributes…
    - ***Simple***: Each entity has a single atomic value
        - Ex. SSN, Sex
    - ***Composite***: The attribute may be composed of several components
        - Ex. Name(FirstName, MiddleName, LastName)
    - ***Multi-Valued***: An entity may have multiple values for that attribute
        - Ex. Color of a CAR, PreviousDegrees of a STUDENT
    - NOTE: A specific entity will have a value for each of its attributes, where each attribute has a *value set*/data type associated with it

Entity Types and Key Attributes

- DEFINITION: Entities with the same basic attributes are grouped/typed into an **entity type**
    - Ex. the entity type EMPLOYEE and PROJECT
- DEFINITION: An attribute of an entity type for which each entity *must* have a unique value is called a **key attribute** of the entity type
    - Ex. SSN of EMPLOYEE
    - NOTE: A key attribute may be composite
        - Ex. VehicleTagNumber is a key of the CAR entity type with components (Number, State)
    - NOTE: An entity type may have more than one key
        - Ex. The CAR entity type may have two keys:
            - VehicleIdentificationNumber (VIN)
            - VehicleTagNumber (Number, State), aka license plate number
    - In the ER diagram, key attributes are <u>underlined</u>

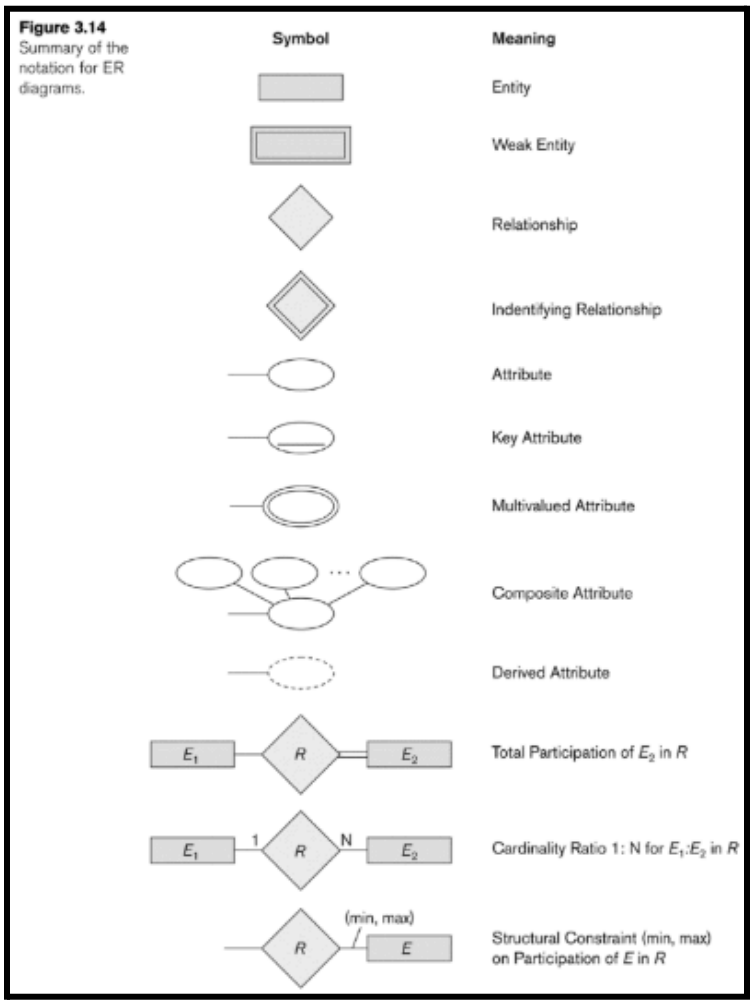
Entity Set

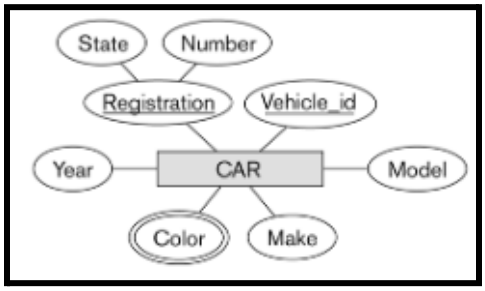
- DEFINITION: Each entity type will have a collection of entities stored in the database called the **entity set** (**entity collection**)
- The entity type and entity set may be given different names
    - Ex. PERSON could be the entity type and POPULATION could be the entity set
- The entity set is the current *state* of the entities of that type that are stored in the database

Value Sets (Domains) of Attributes

- DEFINITION: A **value set** specifies the set of values associated with an attribute
    - Each simple attribute is associated with a *value set*
        - Ex. DOB is a Date YYYY-MM-DD
        - Ex. Lastname is a value which is a character string up to 15 characters

Displaying an Entity Type





Module 7: Relationship, Cardinality & Participation Concepts

- This module is an extension of the previous module, Module 6

Relationships in ER Diagrams

- An ER model has three main concepts…
  - *Entities* (and their entity types and entity sets)
  - *Attributes* (simple, composite, multivalued)
  - ***Relationships*** (and their relationship types and relationship sets)
- DEFINITION: A **relationship** relates two or more distinct entities with a specific meaning
  - Ex. EMPLOYEE John Smith *works on* ProductX PROJECT
- DEFINITION: Relationships of the same type are grouped or typed into a **relationship type**
  - Ex. The WORKS_ON relationship type in which EMPLOYEEs and PROJECTs participate
- The degree of a relationship type is the number of participating entity types
  - Ex. The WORKS_ON relationship is a *binary* relationship because there are only two participating entities

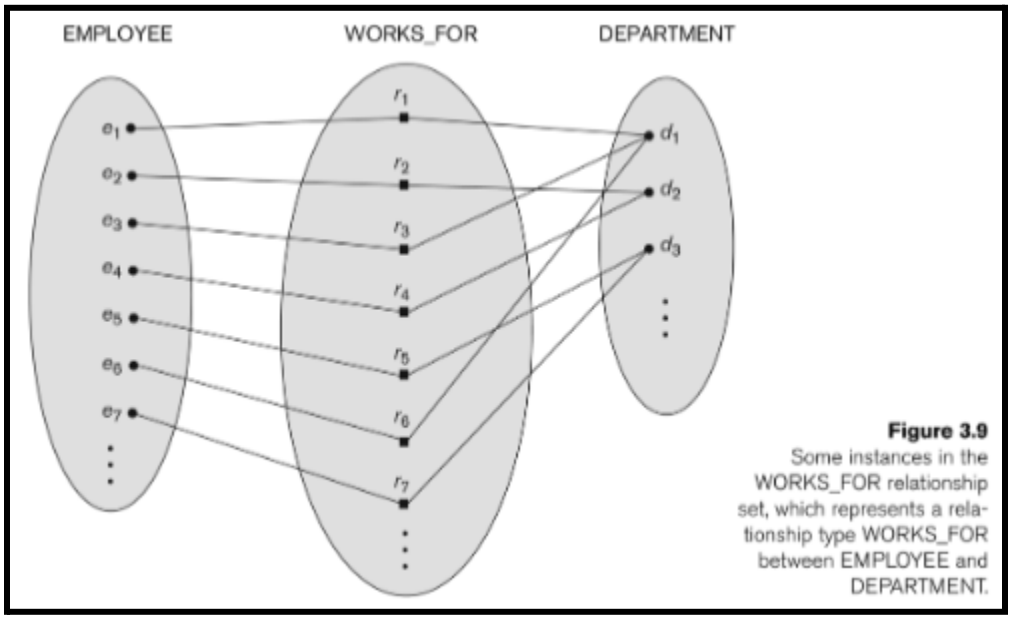


FIGURE: The above displays the relationship set for WORKS_FOR

Relationship Type v. Relationship Set

- A relationship type is the schema description of a relationship; it identifies the relationship name and the participating entities
- DEFINITION: A **relationship set** is the current set of relationship instances represented in the database; the current *state* of a relationship type

- In ER diagrams, we represent the *relationship type* as follows…
    - Diamond-shaped box is used to display a relationship type
    - Connected to the participating entity types via straight lines
    - Note that the relationship type is not shown with an arrow. The name should be read from left to right and top to bottom of your diagram

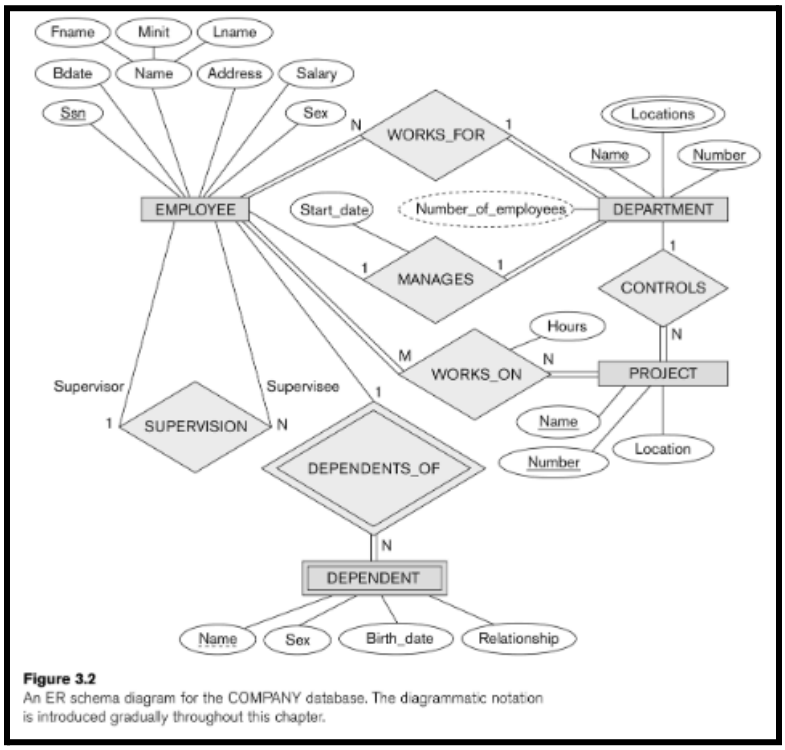
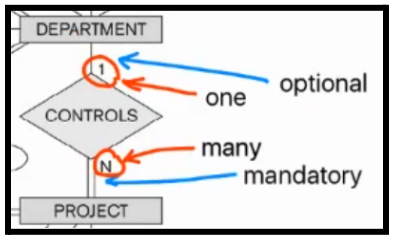


Figure 3.2
An ER schema diagram for the COMPANY database. The diagrammatic notation is introduced gradually throughout this chapter.

- More than one relationship type can exist between the same participating entities
    - Ex. MANAGES and WORKS_FOR are distinct relationship types between EMPLOYEE and DEPARTMENT in the figure above

Constraints on Relationships

- DEFINITION: Each relationship has a **cardinality ratio** (*ratio constraint*) which specifies the *maximum* participation in a relationship. They are as follows…
    - One-to-One (1:1)
    - One-to-many (1:N) or Many to one (N:1)
    - Many to many (M:N)
- DEFINITION: **Existence dependency constraint** (*participation constraint*) specifies the *minimum* participation. They are as follows…
    - zero (optional participation, not existence-dependent)
        - Represented with a single line to the relationship
    - one or more (mandatory participation, existence-dependent)
        - Represented with a double line to the relationship



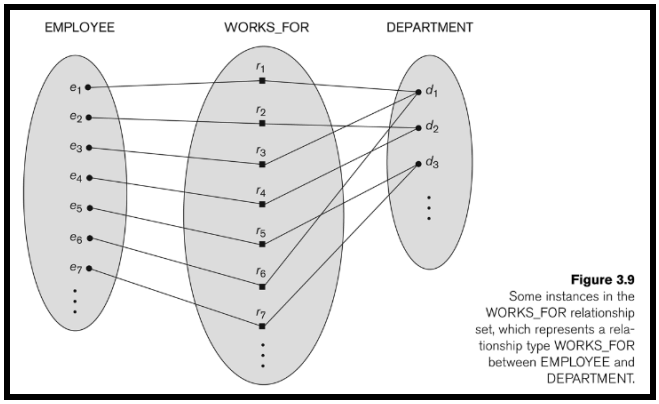


Figure 3.9
Some instances in the WORKS_FOR relationship set, which represents a relationship type WORKS_FOR between EMPLOYEE and DEPARTMENT.

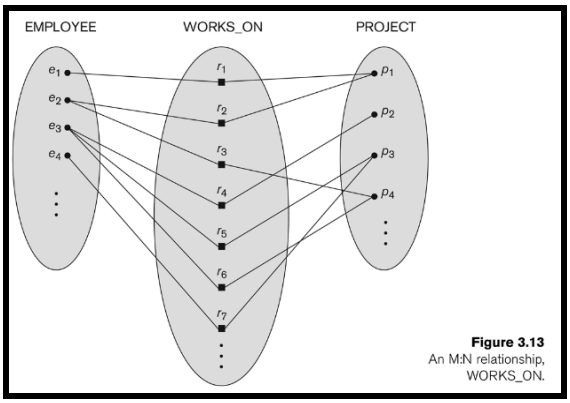


Figure 3.13
An M:N relationship, WORKS_ON.

Derived Attributes

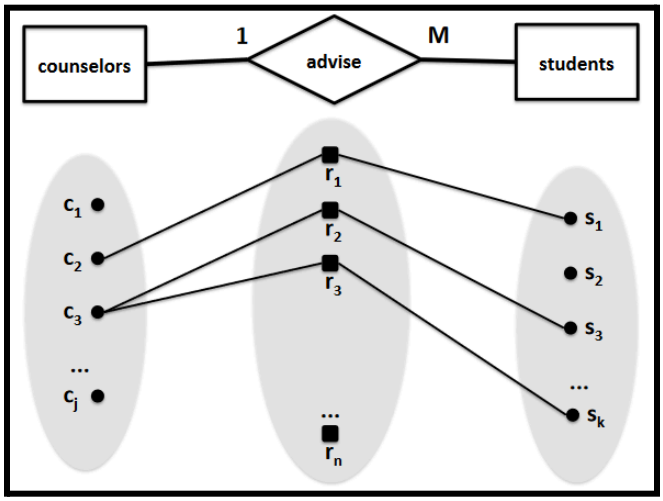
- DEFINITION: A **derived attribute** is an attribute which isn't kept track of in a database, but is rather calculated/derived from data elsewhere in our database
    - Ex. We have EMPLOYEE entities and want DEPARTMENT to keep track of the number of EMPLOYEES in a DEPARTMENT. We can calculate it by querying our EMPLOYEEs and counting the number which work in a DEPARTMENT
    - NOTE: This does not have to take the form of a query. It could also be a mathematical equation and more
- Derived attributes are represented with a dotted oval connected to an entity

Module 9: Calculating the Minimum & Maximum Number of Relationships

- Note that Module 8 is a overview of Phase I of the project, and as such won't be listed here
- There are two aspects of relationships that are of interest to us…
    - Total v. Partial participation
    - Relationship cardinality

Cardinality

- Talks about *how many* things you can be involved with in another set
- Represented as one of the three…
    - One-to-One (1:1)
    - One-to-many (1:N) or Many to one (N:1)
    - Many to many (M:N)
- NOTE: This does **not** mean that they are *required* to be involved with an entity of another set. That is reserved for total v. partial participation.

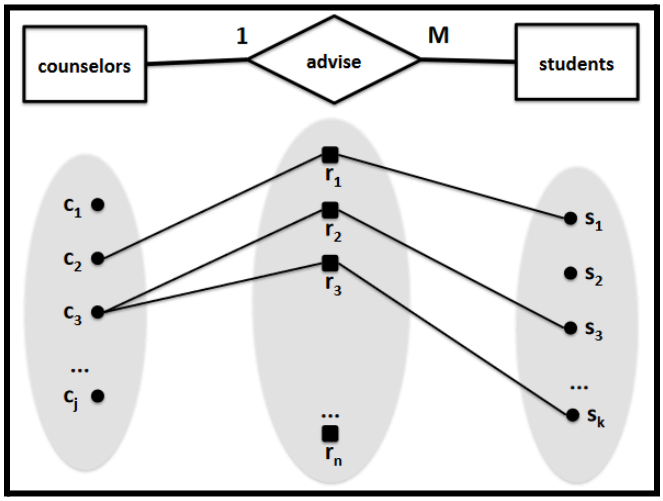


- In the above figure, we have a 1:M relationship. Here is how this would be read in English…
    - A counselor can advise *many* students (it can be 0, it can be 1, it can be 2, etc.)
    - A student can be advised by *at most* one counselor (the student *may or may not* have one counselor)

Participation

- Talks about requirements for being involved with another set
- Represented as one of the two…
    - zero (optional participation, not existence-dependent)
        - Represented with a single line to the relationship
    - one or more (mandatory participation, existence-dependent)

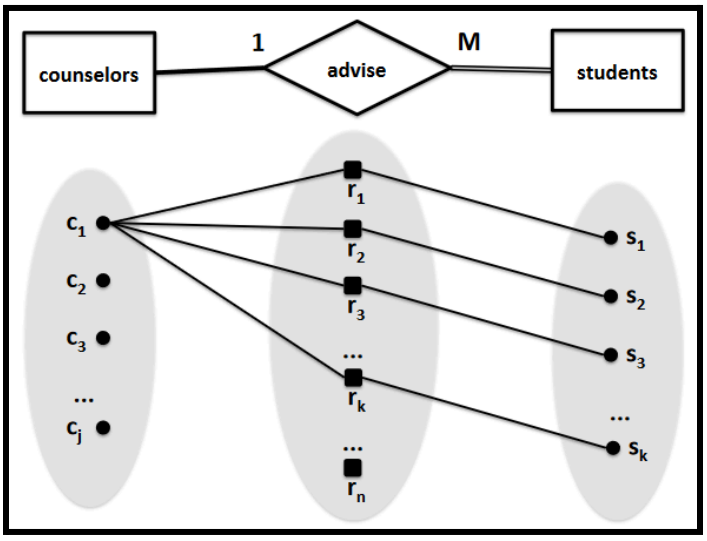
- Represented with a double line to the relationship



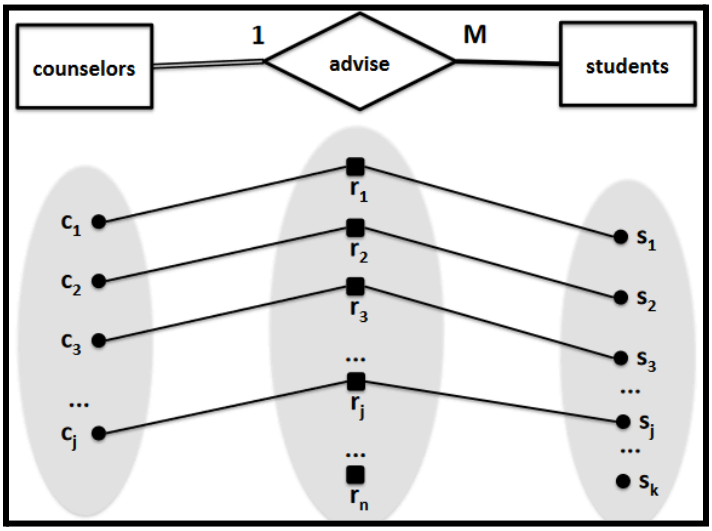
- In the above figure, we have optional participation for both counselors and students. Here is how this would be read in English…
  - A counselor *can* advise many students (it can be 0, it can be 1, it can be 2, etc.)
  - A student *can* be advised by at most one counselor (the student *may or may not* have one counselor)
    - NOTE: Pay attention to the different italicized words between the two examples
- Partial participation is represented by a single line connected to the relationship; total participation is represented by a double line connected to the relationship

Estimating the Size of the Relation Between two Entities

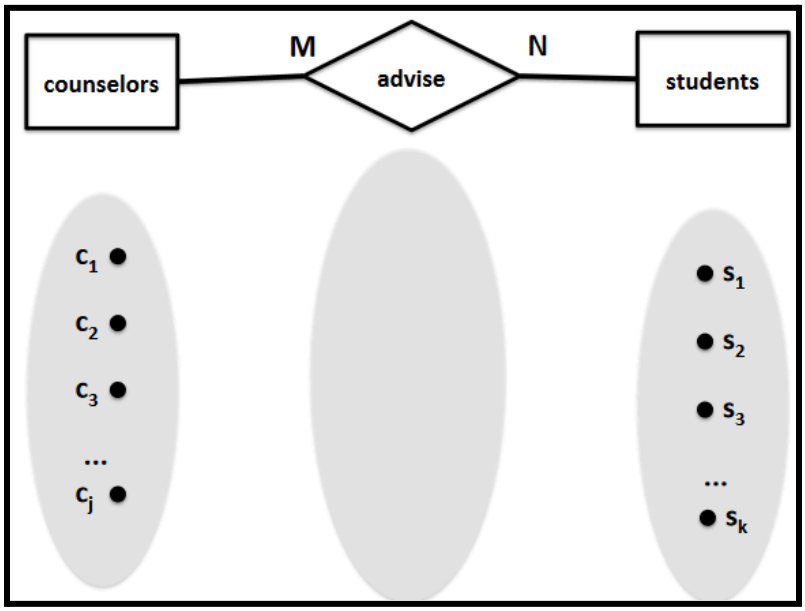
- If you want to determine the *minimum* value, look at the total or partial *participation* of each entity
- If you want to determine the *maximum* value, look at the *cardinality* for each entity
- Examples…



- What is the *minimum* number of relationships?
  - Every student must participate, so your minimum number of relationships is the number of students you have, which is *k*
- What is the *maximum* number of relationships?
  - Every student must have one counselor, so your max number of relationships is the number of students you have, which is *k*



- What is the *minimum* number of relationships?
  - Every counselor must participate, so your minimum number of relationships is the number of counselors you have, which is *j*
- What is the *maximum* number of relationships?
  - Every counselor can advise at most M students, so if every counselor accounted for every student then your max number of relationships is the number of students you have, which is *k*

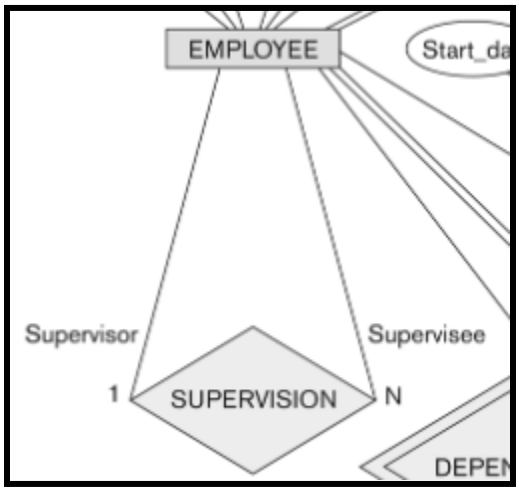


- What is the *minimum* number of relationships?
  - Since no one is required to advise or be advised, the minimum number of relationships is 0
- What is the *maximum* number of relationships?
  - We have *j* counselors and *k* students. Since M counselors can advise N students, if every counselor advised every student, then our number of relationships would be *k*j*

Module 10: Enhanced and Advanced ERD Topics

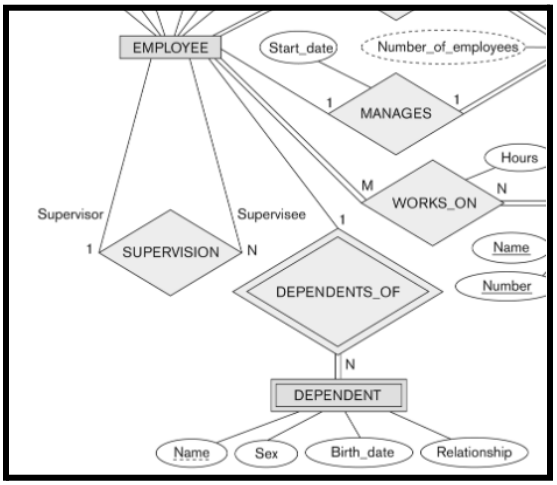
Recursive Relationship Type

- DEFINITION: A **recursive (self-referencing) relationship** is a relationship type between the same participating entity type in *distinct* roles
    - Ex. the SUPERVISION relationship; an employee can both be the supervisor and the supervisee
    - Each relationship instance relates two distinct EMPLOYEE entities, one in the supervisor role and another in the supervisee role
- In an ER diagram, you need to display the role names to distinguish participations



Weak Entity Types

- DEFINITION: A **weak entity** is an entity that does not have a key attribute and that is identification-dependent on another entity type
    - As such, a weak entity *must* participate in an identifying relationship type with an owner or identifying entity type
- Weak entities are identified by the combination of…
    - A *partial key* of the weak entity type, and
    - The particular entity they are related to in the identifying relationship type
- In an ER diagram…
    - A double-lined box identifies a *weak entity*
    - A double-lined diamond is an *identifying relationship*
    - A dash-underlined attribute is a *partial (discriminant) key*



Attributes of Relationship Types

- A relationship type can have attributes
    - Ex. HoursPerWeek of WORKS_ON, indicating how many hours per week an EMPLOYEE works on a PROJECT
- Most relationship attributes are used with M:N relationships
    - In 1:N relationships, they can be transferred to the entity type on the N-side of the relationship

Enhanced Entity-Relationship Models: Subclasses & Superclasses

- It's entirely possible that an entity type may have additional meaningful subgroupings of its entities
    - Ex. EMPLOYEE can be further broken down into…
        - MANAGER
        - SALARIED_EMPLOYEE, HOURLY_EMPLOYEE
        - etc.
- EER diagrams extend ER diagrams to represent these additional subgroupings called *subclasses* or *subtypes*

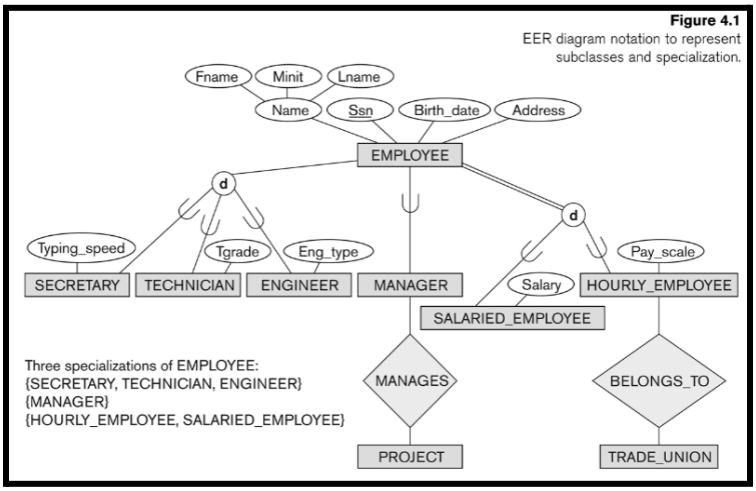


FIGURE: Each subgrouping is a subset of EMPLOYEE entities, and each is called a *subclass* of EMPLOYEE. Inversely, EMPLOYEE is the *superclass* of all the subclasses

- This kind of hierarchy is sometimes referred to as IS-A relationships
    - Ex. SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE, etc.
    - Because of this logic, a *subclass* doesn't exist without it's *superclass*, since a *subclass* is defined by the *superclass*
- NOTE: It is not necessary that every entity in a superclass be a member of some subclass. In the first figure, the designers made this the case by having mandatory participation with the disjoint, however it is possible to have both an EMPLOYEEs and SECRETARYs
- An entity that is a member of a subclass *inherits*…
    - All attributes of the entity as a member of the superclass
    - All relationships of the entity as a member of the superclass

Enhanced Entity-Relationship Models: Specialization

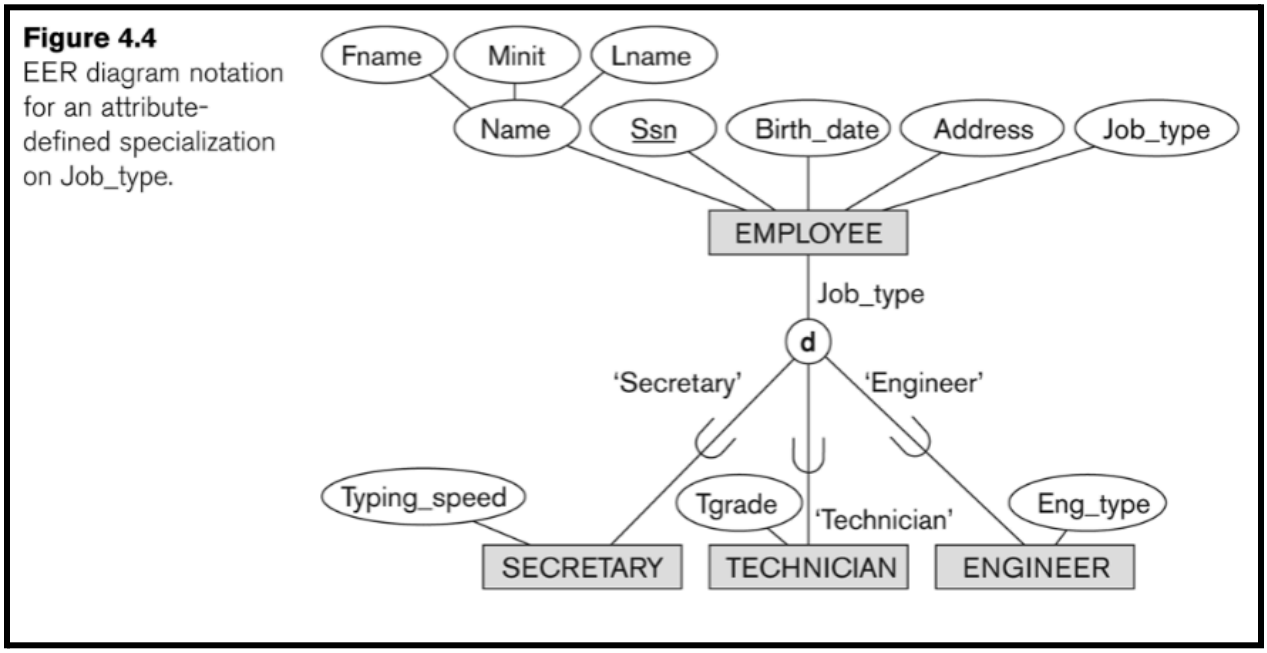
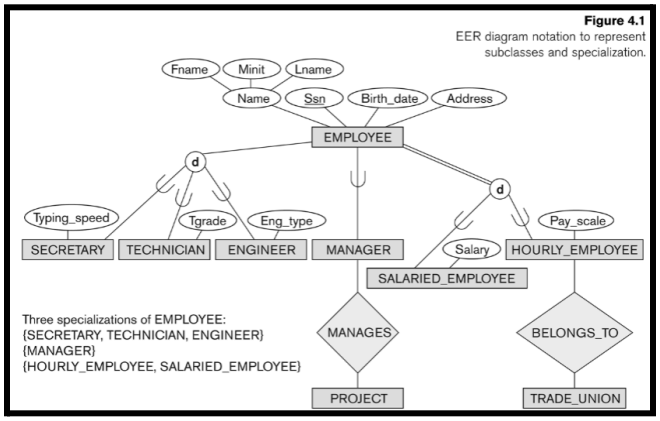


FIGURE: In the above figure, the EMPLOYEE superclass specializes into subclasses. Note that each line has text explicitly indicating this specialization

- DEFINITION: **Specialization** is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
    - Ex. MANAGER is a specialization of EMPLOYEE based on the role the employee plays
- Attributes of a subclass are called *specific* or *local* attributes, because they only pertain to that subclass and not the superclass
    - Ex. Typing_speed of SECRETARY is a *local* attribute
- Subclasses can participate in specific relationship types just like any other entity
    - Ex. The relationship BELONGS_TO of HOURLY_EMPLOYEE



Enhanced Entity-Relationship Models: Generalization

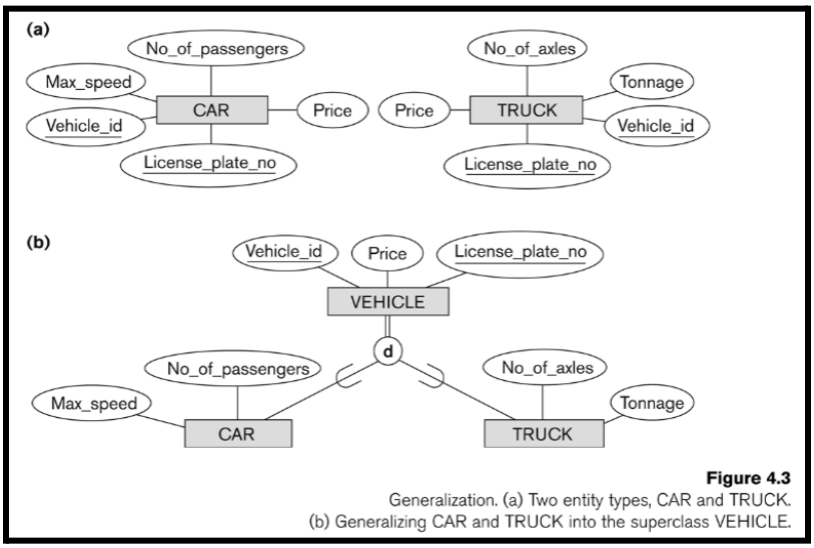


Figure 4.3
Generalization. (a) Two entity types, CAR and TRUCK. (b) Generalizing CAR and TRUCK into the superclass VEHICLE.

FIGURE: We can use a *subset* symbol on a line to indicate that an entity is a *sub*class of another entity. We can also use a circle with a *d* inside of it to represent disjoint. In this above example, VEHICLE has a mandatory participation to this disjoint, meaning all vehicles in the database *must* be either a CAR or TRUCK. In other words, VEHICLE is *abstract*.

- DEFINITION: **Generalization** is the reverse of the specialization process; several classes with common features are generalized into a superclass
    - Original classes become subclasses
    - Ex. CAR, TRUCK were generalized into VEHICLE
        - {CAR, TRUCK} became *specializations* of VEHICLE

Constraints on Specialization and Generalization

- There are two basic constraints which can apply to specialization and generalization
    - *Disjointness*
    - *Completeness*
- Disjointness
    - DEFINITION: The **disjointness constraint** specifies that the subclass of the specialization must be *disjoint*. That is, an entity can be a member of at most one of the subclasses of the specialization (typical OOP inheritance)
    - Specified with a *d* in the EER diagram
    - If not disjoint, specialization is *overlapping*. That is, the same entity may be a member of more than one subclass of the specialization
        - Specified with an *o* in the EER diagram

- Completeness
  - DEFINITION: The **completeness (exhaustiveness) constraint** specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization
  - Shown in EER diagrams by a double line
  - If not complete, then we are partial. That is, an entity is not required to belong to any of the subclasses
    - Shown in EER diagrams by a single line

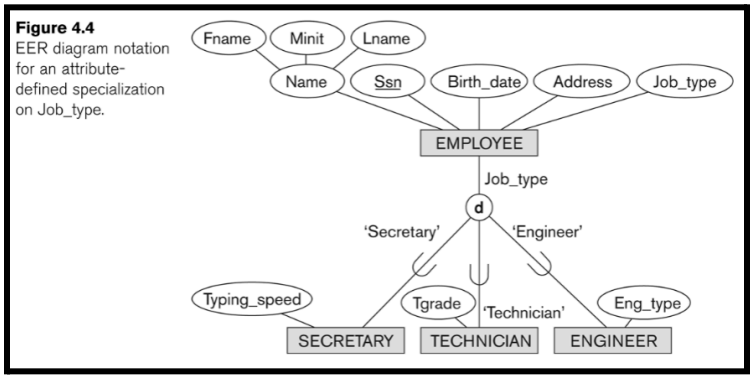


FIGURE: The above uses disjoint, partial specialization/generalization

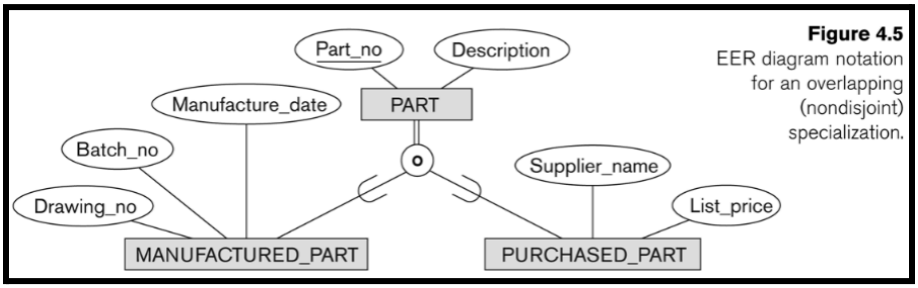
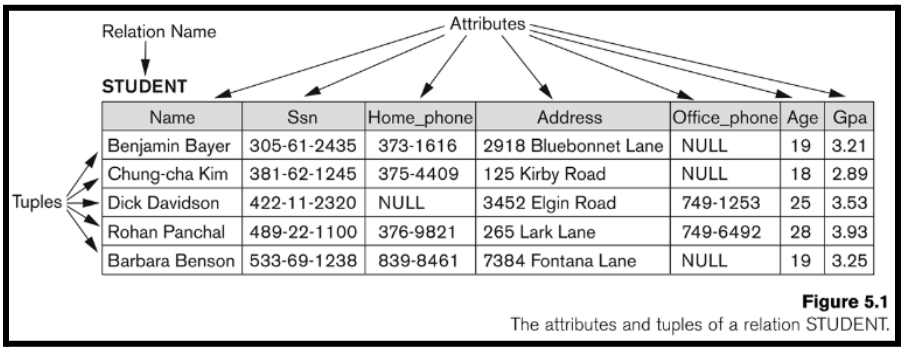


FIGURE: The above uses overlapping, total specialization/generalization

Module 11: Key & Entity Integrity Constraints

Formal Relational Model

- A *relation* is a mathematical concept based on the ideas of sets
- Informally, we can say that a *relation* looks like a *table* of values containing a set of rows
- We've said that the data elements in each row represent certain facts that correspond to a real-world *entity* or *relationship*
    - In the formal model, we call rows **tuples**
- Each column had a column header that gave an indication of the meaning of the data items in that column
    - In the formal model, we call column headers **attributes**



Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Figure 5.1
The attributes and tuples of a relation STUDENT.

- DEFINITION: A **key** is a value of a data item (or set of items) that uniquely identifies the row in the table
    - Ex. In the STUDENT table, SSN is the key
- Sometimes row-ids or sequential numbers are assigned as keys to identify the rows in a table; These are called *artificial* (*surrogate*) *keys*

Formal Definitions

- Schema
    - The **schema** (or description) of a relation is denoted by $R(A_1, A_2, \ldots, A_n)$, where R is the *name* of the relation and the *attributes* of the relation are $A_1, A_2, \ldots, A_n$
        - Ex. CUSTOMER(Id, Name, Address, Phone#)
        - CUSTOMER is the relation name defined over four attributes: Id, Name, Address, Phone#
    - Each attribute has a *value set* (*domain*)
        - Ex. Domain of Id is 6 digit numbers
        - NOTE: This is not to be confused with the `DOMAIN` keyword in SQL
- Tuple
    - A **tuple** is an ordered set of values, depicted with values enclosed in angled brackets '< … >'
    - Each value in the tuple must be derived from an appropriate *domain*

        - Ex. A row in the CUSTOMER relation is a 4-tuple…
        - <632895, "John Smith", "101 Main St. Atlanta, GA 30332",
        - "(404) 894-2000">
    - A **relation** is a set of such tuples (rows)
    - All values in a tuple are considered *atomic* (indivisible)
    - A special **null** value is used to represent values that are unknown or not available or inapplicable in certain tuples
    - Component values of a tuple *t* are referred to with the following notation: $t[A_i]$ or t.Ai
        - This is value v_i of attribute A_i for tuple *t*
        - Subtuples can be created using $t[A_u, A_v, \ldots, A_w]$
- Domain
    - A **domain** specifies the set of values associated with a value in a tuple
        - Ex. USA_phone_numbers are the set of 10 digit phone numbers valid in the US
    - Domains have a data type or format defined for it
        - Ex. USA_phone_numbers have the format ddd-ddd-dddd where each d is an integer digit
    - The attribute name designates the role played by a domain in a relation
        - Ex. the domain Date may be used to define two attributes named "Invoice-date" and "Payment-date" with different meanings
- State
    - The **relational state** is a subset of the Cartesian product of the domains of its attributes
        - Each domain contains the set of all possible values the attribute can take
        - Ex. Attribute Cust-name is defined over the domain of character strings of maximum length 25
        - dom(`Cust-name`) is `varchar(25)`
        - The role these strings play in the CUSTOMER relation is that of the *name of a customer*
    - The state of a schema is given by r(R), and outputs a set of tuples
        - $r(R) = \{t_1, t_2, \ldots, t_n\}$, where each t_i is an *n*-tuple
        - $t_i =< v_1, v_2, \ldots, v_n >$, where each v_j is an element of $dom(A_j)$
    - NOTE: The tuples in a relation r(R) are not considered to be ordered, even though they appear to be in the tabular form
- Example
    - Let $R(A_1, A_2)$ be the relation schema
        - Let $dom(A_1) = \{0, 1\}$

Informal Terms		Formal Terms
Table		Relation
Column Header		Attribute
All possible Column Values		Domain
Row		Tuple
Table Definition		Schema of a Relation
Populated Table		State of the Relation

        - Let dom(A_2) = {a, b, c}
    - Then, dom(A_1) X dom(A_2) is all possible combinations
        - {< 0, a > , < 0, b > , < 0, c >, < 1, a >, < 1, b >, < 1, c > }
    - The relation state r(R) is a subset of this Cartesian product
        - $r(R) \subset dom(A_1)\ X\ dom(A_2)$

Constraints

- DEFINITION: **Constraints** determine which values are permissible and which are not in a database. There are three types…
    - *Inherent* (*Implicit*): Based on the data model itself
        - e.g. relational model does now allow a list as a value for any attribute
    - *Schema-based* (*Explicit*): Expressed in the schema by using the facilities provided by the model
        - e.g. maximum cardinality ratio constraint in the ER model
    - *Application-based* (*Semantic*): These are beyond the expressive power of the model and must be specified and enforced by the application programs
- Constraints are *conditions* that must hold on all valid relation states
- There are three main types of explicit constraints that can be expressed in the relational model…
    - **Key** constraints
    - **Entity integrity** constraints
    - **Referential integrity** constraints
- Another schema-based constraint is the **domain** constraint, which states that every value in a tuple must be from the domain of its attribute (or null, if allowed for that attribute)

Key Constraints

- DEFINITION: A **superkey** of R is a set of attributes S_k of R with the following condition
    - No two tuples in any valid relation state r(R) will have the same value for S_k
        - That is, for any distinct tuples t_1 and t_2 in r(R), $t_1[S_k]$!= $t_2[S_k]$
- DEFINITION: A **key** of R is a superkey K such that removal of any attribute from K results in a set of attributes that does not possess the superkey uniqueness property
    - A key is a superkey but not vice versa
- Example
    - Consider the CAR relation schema CAR(State, Reg#, SerialNo, Make, Model, Year)
        - CAR has two keys…
            - Key1 = {State, Reg#}
            - Key2 = SerialNo}
        - Both are superkeys of CAR
        - {SerialNo, Make} is a superkey but *not* a key

- In general…
  - Any key is a *superkey* but not vice versa
  - Any set of attributes that *includes* a key is a *superkey*
  - A *minimal* superkey is also a key
- DEFINITION: If a relation has several *candidate keys*, one is chosen arbitrarily to be the **primary key**
  - Primary key attributes are underlined
  - Ex. In our CAR relation schema, we choose SerialNo to be the primary key → CAR(State, Reg#, SerialNo, Make, Model, Year)
- The primary key value is used to uniquely identify each tuple in a relation; provides the tuple identity
- In general, the primary key should be the smallest candidate key in terms of size (but not always)

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Figure 5.4
The CAR relation, with two candidate keys: License_number and Engine_serial_number.

Relational Database Schema

- DEFINITION: **Relational database schema** is a set S of relation schemas that belong to the same database
- S is the name of the whole database schema
- S = {R_1, R_2, …, R_n}, where R_1, R_2, …, R_n are the names of individual relation schemas within the database S

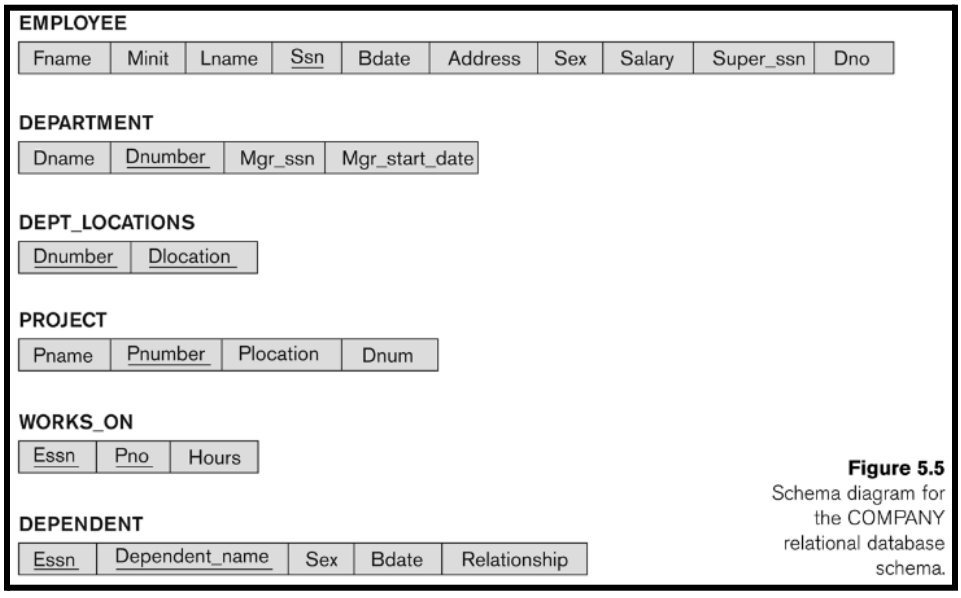


Figure 5.5
Schema diagram for the COMPANY relational database schema.

Relational Database State

- DEFINITION: A **relational database state** DB of S is a set of relation states DB = {r_1, r_2, …, r_m} such that each r_i is a state R_i and such that the r_i relation states satisfy the integrity constraints specified
    - Sometimes called a relational database *snapshot* or *instance*
- Whenever the database is changed, a new state is made
    - INSERT, DELETE, and MODIFY operations all change the state of the database

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Entity Integrity

- DEFINITION: **Entity integrity** is a constraint that states that the primary key attributes PK of each relation schema R in S cannot have null values in any tuple of r(R)
    - This is because primary key values are used to *identify* the individual tuples
    - If PK has several attributes, null is not allowed in *any* of the attributes
- NOTE: Other attributes of R may be constrained to disallow null values, even though they are not members of the primary key

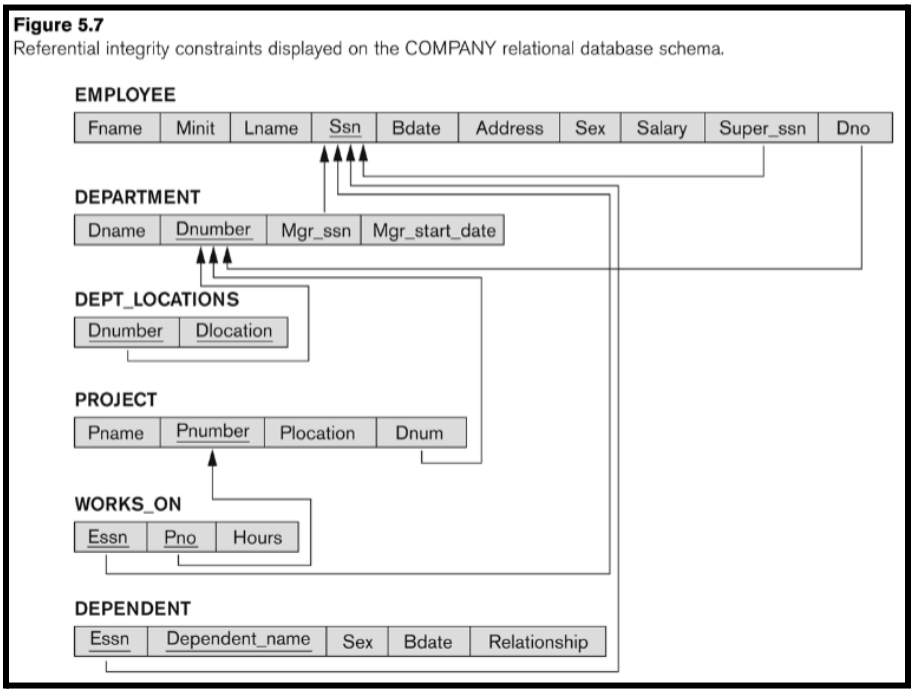
Module 12: Referential Integrity Constraints

Referential Integrity

- Our previous constraints involved one relation; referential integrity involves two relations, and is used to specify a *relationship* among tuples in two relations
    - Called the *referencing relation* and the referenced relation
- Tuples in the *referencing relation* R_1 have attributes FK called **foreign key** attributes that reference the primary key attributes PK of the referenced relation R_2
    - A tuple t_1 in R_1 is said to reference a tuple t_2 in R2 if $t_1[FK] = t_2[PK]$
- DEFINITION: **Referential integrity** is a constraint where the value in the foreign key column (or columns) FK of the *referencing relation* R_1 can be either…
    - (1) a value of an existing primary key of a corresponding primary key PK in the referenced relation R_2, or
    - (2) a null
- NOTE: In case 2, the FK in R_1 should not be a part of its own primary key

Displaying a Relational Database Schema and Its Constraints

- Each relation schema can be displayed as a row of attribute names
- The name of the relation is written above the attribute names
- The primary key attribute(s) will be underlined
- A foreign key (referential integrity) constraints is displayed as a directed arc (arrow) from the foreign key attributes to the referenced table or point to the primary key of the referenced relation



Miscellaneous Constraints

- Semantic Integrity Constraints
    - Based on application semantics and cannot be expressed by the model per se
    - Ex. The max number of hours per employee for all projects he or she works on is 56 hours per week

Update Operations on Relations

- Integrity constraints should not be violated by the update operations INSERT, DELETE, or MODIFY
- You can *propagate* updates to cause other updates automatically, which may be necessary to maintain integrity constraints
- INSERT may violate the following…
    - Domain Constraint
        - If one of the attribute values provided for the new tuple is not of the specified attribute domain
    - Key Constraint
        - If the value of a key attribute in the new tuple already exists in another tuple in the relation
    - Referential Integrity
        - If a foreign key value in the new tuple references a primary key value that does not exist in the referenced relation
    - Entity Integrity
        - If the primary key value is null in the new tuple
- DELETE may violate the following…
    - Referential Integrity
        - If the primary key value of the tuple being deleted is referenced from other tuples in the database
            - Can be remedied using one of three actions
                - RESTRICT option: reject the deletion
                - CASCADE option: propagate the new primary key value into the foreign keys of the referencing tuples
                - SET NULL option: set the foreign keys of the referencing tuples to NULL
- UPDATE may violate the following…
    - Domain Constraint
        - If the new attribute value provided is not of the specified attribute domain
- UPDATE may also violate the other constraints depending on what's being updated…
    - Updating the primary key (PK)
        - Similar to a DELETE followed by an INSERT
    - Updating a foreign key (FK)

        - May violate referential integrity
    - Updating an ordinary attribute (neither PK or FK)
        - Can only violate domain constraints

DELETE

- Removes tuples from a relation; removes based on the WHERE-clause…
    - Included clause: selects all tuples which satisfy the WHERE clause
    - Not included clause: specifies that all tuples in the relation are to be deleted
- Referential integrity should be enforced
- Tuples are deleted from only one table at a time (unless CASCADE is specified on a referential integrity constraint)

```
U4A:  DELETE FROM  EMPLOYEE
      WHERE        Lname='Brown';

U4B:  DELETE FROM  EMPLOYEE
      WHERE        Ssn='123456789';

U4C:  DELETE FROM  EMPLOYEE
      WHERE        Dno=5;

U4D:  DELETE FROM  EMPLOYEE;
```

UPDATE

- Used to modify attribute values of one or more selected tuples; uses a WHERE-clause to select the tuples to be modified
- An additional SET-clause specifies the attributes to be modified and their new values
- Referential integrity specified as part of the DDL specification is enforced

```
U5:   UPDATE   PROJECT
      SET      PLOCATION = 'Bellaire',
                   DNUM = 5
      WHERE    PNUMBER=10
```

Module 13: Schema Mapping Exercise

ER-to-Relational Mapping Algorithm

- 1) Mapping of Regular Entity Types
- 2) Mapping of Weak Entity Types
- 3) Mapping of Binary 1:1 Relation Types
- 4) Mapping of Binary 1:N Relationship Types
- 5) Mapping of Binary M:N Relationship Types
- 6) Mapping of Multivalued Attributes
- 7) Mapping of N-ary Relationship Types (not really relevant to us)

Mapping EER Model Constructs to Relations

- 8) Options for Mapping Specialization or Generalization
- 9) Mapping of Union Types (Categories)

Step 1: Mapping of Regular Entity Types

- For each regular entity type E in the ER schema…
    - Create a relation R that includes all the simple attributes of E
    - Choose one of the key attributes of E as the primary key of R
        - If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R

Step 2: Mapping of Weak Entity Types

- For each weak entity type W in the ER schema with owner entity type E…
    - Create a relation R & include all simple attributes (or simple components of composite attributes) of W as attributes of R
    - Include as foreign key attributes of R the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s)
        - So, the primary key of R becomes the *combination of* the primary key(s) of the owner(s) and the partial key of the weak entity type W, if any

Step 3: Mapping of Binary 1:1 Relation Types

- For each binary 1:1 relationship type R in the ER schema…
    - Identify the relations S and T that correspond to the entity types participating in R and do one of three things..
        - 1) Foreign Key Approach (2 relations): Choose one of the relations (for example, S) and include a foreign key in S the primary key of T
            - NOTE: It is better to choose an entity type with total participation in R in the role of S. In other words, the entity that has total participation should have the foreign key

        - 2) Merged Relation Option (1 relation): An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation; may be appropriate when both participations are total
        - 3) Cross-Reference or Relationship Relation (3 relations): Set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types

Step 4: Mapping of Binary 1:N Relationship Types

- For each regular binary 1:N relationship type R…
    - Identify the relation S that represent the participating entity type at the N-side of the relationship type
    - Include as a foreign key in S the primary key of the relation T that represents the other entity type participating in R
    - Include any simple attributes of the 1:N relation type as attributes of S

Step 5: Mapping of Binary M:N Relationship Types

- For each regular binary M:N relationship type R…
    - Create a new relation S to represent R (this is a *relationship relation*)
    - Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types; their combination will form the primary key of S
    - Include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of S

Step 6: Mapping of Multivalued Attributes

- For each multivalued attribute A…
    - Create a new relation R
    - This relation R will include an attribute corresponding to A, plus the primary key attribute K (as a foreign key in R) of the relation that represents the entity type of relationship type that has A as an attribute
    - The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components

Step 7: Mapping of N-ary Relationship Types

- Not relevant to the course

Step 8: Options for Mapping Specialization or Generalization

- Convert each specialization with m subclasses $\{S_1, S_2, \ldots, S_m\}$ and generalized superclass C, where the attributes of C are $\{k, a_1, \ldots, a_n\}$ and k is the primary key, into relational schemas using one of the four following options…
    - Option A: Multiple Relations-Superclass and Subclass
        - Create a relation L for C with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\}$ and $PK(L) = k$. Create a relation L_i for each subclass S_i, $1 < i < m$, with the attributes $Attrs(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $PK(L_i) = k$. This options works for any specialization (total or partial, disjoint of over-lapping)
    - Option B: Multiple Relations-Subclass Relations Only
        - Create a relation L_i for each subclass S_i, $1 < i < m$, with the attributes $Attr(L_i) = \{\text{attributes of } S_i\} \cup \{k, a_1, \ldots, a_n\}$ and $PK(L_i) = k$. This option only works for a specialization whose subclasses are "total" (every entity in the superclass must belong to [at least] one of the subclasses)
    - Option C: Single Relation with One Type Attribute
        - Create a single relation L with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\} \cup \{\text{attributes of } S_1\} \cup \ldots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $PK(L) = k$. The attribute t is called a type (or discriminating) attribute that indicates the subclass to which each tuple belongs
    - Option D: Single Relation with Multiple Type Attributes
        - Create a single relation schema L with attributes $Attrs(L) = \{k, a_1, \ldots, a_n\} \cup \{\text{attributes of } S_1\} \cup \ldots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \ldots, ™\}$ and $PK(L) = k$. Each t_i, $1 < l < m$, is a Boolean type attribute indicating whether a tuple belongs to the subclass S_i.

Step 9: Mapping of Categories (Union Types)

- For mapping a category whose defining superclass have different keys, it is customary to specify a new key attribute, called a *surrogate key*, when creating a relation to correspond to the category

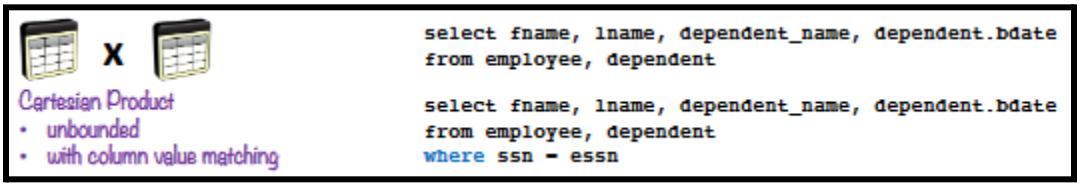
Module 14: Cartesian Products and Inner, Outer & Natural Joins in SQL

- In SQL, you can join tables together in order to create new tables with combined data
    - Suppose you have table A with data *a* and table B with data *b*. SQL provides commands to create a table C with data *a* + *b* (it's not *really* plus, more on that below)
- There are four types of joins we'll see…
- Cartesian Product

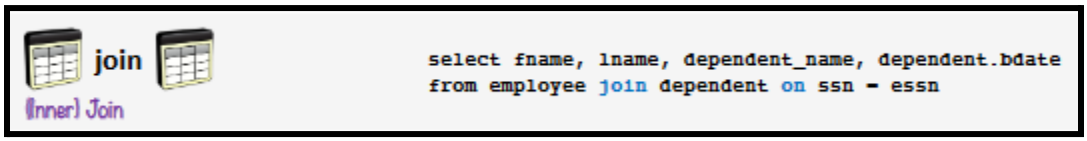
$$(a + b) * (a + b) = a * a + a * b + b * a + b * b = a^2 + ab + ba + b^2 = a^2 + 2ab + b^2$$

$$(a + b + c) * (d + e) = a * d + a * e + b * d + b * e + c * d + c * e$$

    - The *cartesian product* is essentially the exact same as the *distributive property* in algebra
    - This join works by taking every tuple in one table and every tuple in another, and creating a new table which contains every single unique combination of tuples
        - Ex. Suppose table A has tuples $\{t_{1A}, t_{2A}, \ldots, t_{nA}\}$ and table B has tuples $\{t_{1B}, t_{2B}, \ldots, t_{nB}\}$. The cartesian product would create a table C with new tuples $\{t_{1A}t_{1B}, t_{1A}t_{2B}, \ldots, t_{1A}t_{nB}, t_{2A}t_{1B}, t_{2A}t_{2B}, \ldots\}$

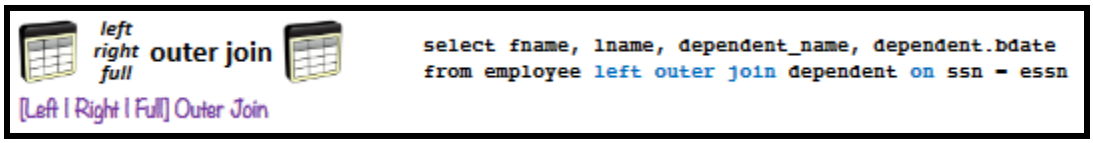


- Join Command
    - Instead of using the *cartesian product* method, you can also use the join command method. Doing this, we can both merge the tables and also use the `where` command to narrow down our table created with the join command.
        - In other words, we use the `join` command to create our table A + B, then use the where command to narrow down our search, as opposed to simply using the where command to generate A + B
    - Doing it this way, you get essentially the exact same results as the cartesian product using a *where* clause, but you save yourself from having to waste the where clause on the join and instead can use the where clause to narrow down the join.

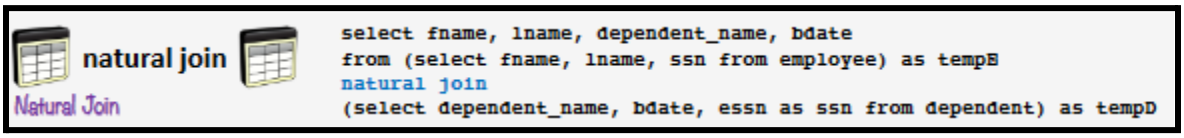


- (Left, Right, Full) Outer Join Command

- Tuple is included in the result only if a matching tuple exists in the other relation
- *Left Outer Join*
  - Every tuple in the left table must appear in result
  - If no matching tuple exists, it's padded with NULL values for attributes of the right table
- *Right Outer Join*
  - Every tuple in right table must appear in result
  - If no matching tuple exists, it's padded with NULL values for attributes of the left table



- Natural Join Command
  - Compares the name of every attribute in table A to every attribute in table B and creates a new table C where every entry with the same attribute name and the same value is joined into one tuple
    - Ex. If table A has an attribute Dnumber and table B has an attribute Dnumber, then the natural join table C will contain new tuples where all tuples in C will have the same Dnumber in their respective tables



Module 15: Nested Queries & Set Operations in SQL

- Much like how in object-oriented programming it's emphasized that we break down our problems into smaller functions, methods, what have you, we can do a similar sort of breakdown with queries in SQL

Nested Queries

- Let's kick off this module by giving an example of why something like this would be helpful
- Suppose we are working with our employee database example. Write a query to retrieve the minimum salary for all of the employees…

```
select min(salary) from employee;
```

- Let's say this outputs 25000. Now, I want a query that gives me all the data for each employee who earns the minimum salary for all of the employees
- One way you could go about doing this is hard coding the value…

```
select * from employee where salary = 25000;
```

- Now this does work, however there is a problem with it…
- A good SQL query should work even if the *state* of the database changes
    - What does this mean? Well remember we have a database *schema* and a database *state*. A *schema* is sort of like building the outline of a house in wood, or perhaps it's the blueprint of the architecture. It's something that can't really change. A *state*, on the other hand, is a snapshot of what the database looks like at that time. This could be akin to the color of paint on a wall, or whether they use carpet or tile, things that can be easily changed (compared to, like, knocking a wall down in this metaphorical house).
    - Our queries should work so long as the *state* changes, queries are not expected to work if the *schema* changes
    - Suppose someone gets hired for the company and they make $24,999. We'd have to change this query. But then another person is hired and they make $24,998. We'd have to change it again. Our query should be able to work even if the state of our database changes
- Naively, how did we get our 25000 number? We simply did a query to get it. What if we could do a query inside of our query?

```
select * from employee
where salary = (select min(salary) from employee);
```

    - NOTE: Putting the parenthesis around the nested query is important because it indicates that we want to run that query *first*. It's just like algebra; parenthesis come first
- There are some other comparison operators we can use to some value *v*..

- **= ANY** (**= SOME**): Returns true if the value *v* is equal to some value in the set *V*, equivalent to the **IN** operator
  - Can also use >, >=, <, <=, and <> with **ANY** (or **SOME**)
- **ALL**: Value must exceed all values from the nested query

```
SELECT   Lname, Fname
FROM     EMPLOYEE
WHERE    Salary > ALL ( SELECT   Salary
                        FROM     EMPLOYEE
                        WHERE    Dno=5 );
```

Set Operations

- Continuing with our example, suppose we now want to get a table containing both the employee data of the employees with minimum salary **and** maximum salary
- Well, similarly to how we can use things like union in set theory, we also have the **union** operator in SQL…

```
select * from employee
where salary = (select min(salary) from employee)
union
select * from employee
where salary = (select max(salary) from employee);
```

- One point to bring up is the "weakness" of the equals symbol (=). Suppose we did the following query…

```
select * from employee
where salary = (select min(salary) from employee
union
select max(salary) from employee);
```

- The equals symbol *only* works on tables with **one row**
- If we did our query in parenthesis, we would get a table with two rows, so this wouldn't work
- Instead, what if replace our equals sign with the in operator from set theory (analogous to the ∈ symbol)

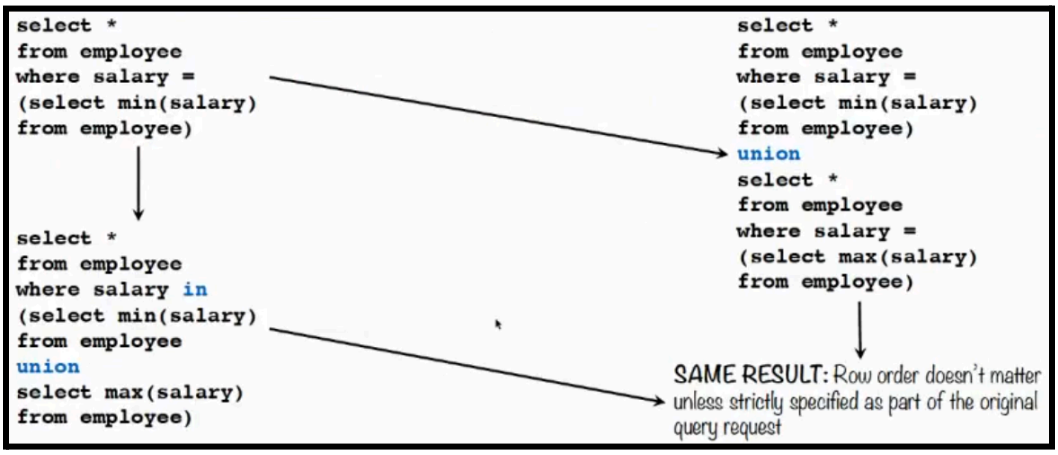
```
select * from employee
where salary in (select min(salary) from employee
union
select max(salary) from employee);
```

- This works too! This is because the *in* operator treats the table as a set, and checks if salary is *in* that set
- Note that another valid query which doesn't use set theory would be the following…

```
select * from employee
where salary = (select min(salary) from employee)
```

```
or salary = (select max(salary) from employee);
```

- You could also switch the equal signs with the **in** operator and get the same answer



- Let's try another query: Retrieve the social security numbers (ssn) for all employees who are either department managers or earn more than 40000
  - If I do **`select ssn from employee where salary > 40000`**, I will get 2 entries
  - If I then do **`select mgrssn from department`**, I get 3 entries
  - But I do **`select ssn from employee where salary > 40000 union select mgrssn from department`**, I only get 3 entries. What's happening here?
    - It's because we have some overlap between the two. In other words, we have entries for people who are both managers and make more than 40000. By default, the **`union`** operator removes duplicates
    - NOTE: If you want to see them everything, including duplicates, you can instead use **`union all`**

The EXISTS and UNIQUE Functions in SQL

- The EXISTS function checks whether the result of a correlated nested query is empty or not; returns a TRUE or FALSE value
  - Can either use EXISTS or NOT EXISTS
- The UNIQUE function returns true if there are no duplicate tuples in the result of query Q

```
SELECT Fname, Lname
FROM Employee
WHERE EXISTS (SELECT *
              FROM DEPENDENT
              WHERE Ssn= Essn)

      AND EXISTS (SELECT  *
                  FROM Department
                  WHERE Ssn= Mgr_Ssn)
```

Module 17: SQL Views

- NOTE: Module 16 had no corresponding notes; it was practice for doing queries, which these notes will not cover.

Views in SQL

- There are two types of tables in SQL…
    - *Base Tables*: These are the tables we are familiar with, and are created using **`CREATE TABLE`**. They hold the raw data that we want to store.
    - *Virtual Tables*: These are tables which are *derived* from other base (or even virtual) tables, otherwise known as **views**, and are created using **`CREATE VIEW`**

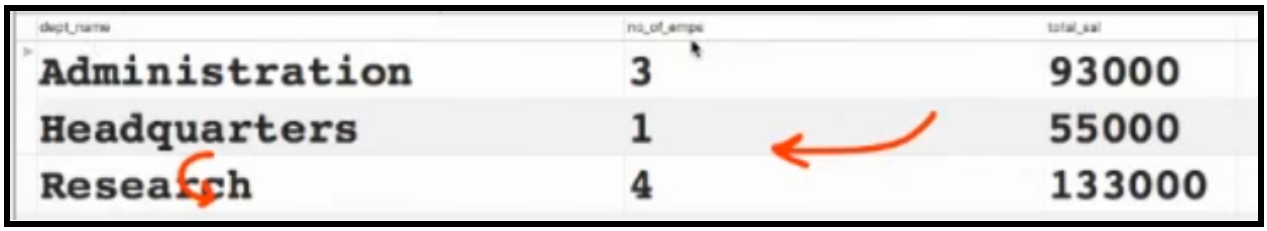
```
V1:  CREATE VIEW   WORKS_ON1
     AS SELECT     Fname, Lname, Pname, Hours
        FROM       EMPLOYEE, PROJECT, WORKS_ON
        WHERE      Ssn=Essn AND Pno=Pnumber;

V2:  CREATE VIEW   DEPT_INFO(Dept_name, No_of_emps, Total_sal)
     AS SELECT     Dname, COUNT (*), SUM (Salary)
        FROM       DEPARTMENT, EMPLOYEE
        WHERE      Dnumber=Dno
        GROUP BY   Dname;
```

- Similar to a base table, once a view is defined it can be used in the FROM clause of an SQL query
- Note that because a view is derived from the information in a base table, views are *always* up-to-date
    - So if the data is changed in the base table, this will be reflected in the view
- The primary difference between a base table and virtual tables are that virtual tables are created through *querying* other tables

```
create view works_on1 as
select fname, lname, pname, hours
from employee, project, works_on
where ssn = essn and pno = pnumber;
```

    - Notice that it doesn't define primary keys, foreign keys, none of that; it's defined by executing a query
- Views are not *created* so much as they are *calculated*--wait, what else do we have that we *calculate*? Derived attributes.
    - So, derived attributes show up in views
- We can also rename the headings of our columns in a view…

```
create view dept_info(dept_name, no_of_emps, total_sal) as
select dname, count(*), sum(salary)
from department, employee
where dnumber = dno group by dname;
```



dept_name	no_of_emps	total_sal
Administration	3	93000
Headquarters	1	55000
Research	4	133000

- It's kind hard to see but you can sort of make out the underscores in their column names

Updating View

- You can update on a view defined on a single table without any aggregate functions
  - Updates are not permitted on aggregate views
  - E.g. The following would not work because Total_sal is an aggregate of all the salaries in that department. If I were to update total_sal, whose salary takes the hit? There's no way to know, so it's disallowed

```
UPDATE    DEPT_INFO
SET       Total_sal=100000
WHERE     Dname='Research';
```

- On views involving joins, it's often not possible

Views as Authorization Mechanism

- Views can be used to hide certain attributes or tuples from unauthorized users
- For instance, for a user who is only allowed to see employee information for those who work for department 5, they may only access the view…

```
CREATE VIEW    DEPT5EMP  AS
SELECT         *
FROM           EMPLOYEE
WHERE          Dno = 5;
```

- We can also ensure that things added into our view follow a strict guideline…

```
create view dept5emp_check as
select * from employee
where dno = 5 with check option;
```

- So now, if we try to insert a value in our view where the tuple's dno is not 5, it will throw an error
  - Note that if you tried to add someone anyway, they simply wouldn't show up, but we shouldn't allow this kind of behavior

Module 18: Functions & Stored Procedures

Functions

```
drop function if exists calc_age;
delimiter //
create function calc_age(birthdate date)
    returns integer deterministic
begin
    return timestampdiff(year, birthdate, current_date());
end //
delimiter ;
```

- Just like in most programming languages, SQL also offers the ability for the user to create functions/methods
- This function, **`calc_age`**, takes in a parameter called **`birthdate`** of type **`date`** and returns the result from **`timestampdiff`**
- To call this specific function, one uses the reserved word **`select`**…

```
select calc_age('1970-05-20');
select calc_age('2000-06-20');
select calc_age('2000-06-22');
```

Stored Procedure

- Based on the term "procedure," you may be inclined to think this is similar to functions
- They are the same in the sense that you abstract an implementation, but instead of *returning* something, you more-so create something (e.g. query)

```
drop procedure if exists direct_workers_fixed;
delimiter //
create procedure direct_workers_fixed()
begin
    select * from employee where superssn = 888665555;
end //
delimiter ;
```

- To call this function, you use the reserved word **`call`**…

```
call direct_workers_fixed();
```

- Notice that this procedure does not contain any variables. That is, it will *always* return the same result provided the data in the database doesn't change; we call these types of procedures *fixed*
- If we want to perform the procedure's query on multiple ssns, then we will need to make this procedure more *flexible* by adding a parameter…

```
drop procedure if exists direct_workers_flexible;
delimiter //
create procedure direct_workers_flexible(in the_ssn decimal(9,0))
begin
    select * from employee where superssn = the_ssn;
end //
delimiter ;
```

- Now, when we call this procedure we call it with parameters…

```
call direct_workers_flexible(888665555);
call direct_workers_flexible(444444444);
call direct_workers_flexible(987654321);
```

What are these Delimiters?

- At this point, you may have seen the delimiter lines in the above figures and question why they are there
- It's actually very important for procedures; notice that the normal delimiter in SQL is a semicolon ;
- If we kept the normal delimiter, then our procedure would end on the first line of our procedure
- However, internally our procedure also needs to know when we are done calling a line internally
- To do this, we create a new delimiter //. This way, in the global view, the procedure ends when we use // but inside the procedure we can also use semicolons ; to indicate when a line is over
- After the procedure, we have to reset the delimiter so we don't break everything else

More on Procedures

- You can also have conditionals inside of your procedures. The way you handle them is similar to how you would handle them in assembly…
- You can also use sets with procedures or use sets as parameters to procedures; this is similar to using a list in Python or an array…

```
drop procedure if exists dept_salary_filter_secure;
delimiter //
create procedure dept_salary_filter_secure(in low_bound integer,
    in high_bound integer, in the_dept integer)
sp_main: begin
    if the_dept = 4 then leave sp_main; end if;

    select * from employee
    where salary between low_bound and high_bound
    and dno = the_dept;
end //
delimiter ;
```

```
drop procedure if exists dept_salary_filter_list;
delimiter //
create procedure dept_salary_filter_list(in low_bound integer,
    in high_bound integer, in list_of_depts varchar(100))
sp_main: begin
    if find_in_set(4, list_of_depts) then leave sp_main; end if;

    select * from employee
    where salary between low_bound and high_bound
    and find_in_set(dno, list_of_depts);
end //
delimiter ;
```

- Procedures can do more than just querying; they can also insert into or update tables

```
drop procedure if exists add_new_dept;
delimiter //
create procedure add_new_dept(in the_name char(20),
    in the_number decimal(1,0), in the_mgr decimal(9,0), in the_start date)
begin
    insert into department values (the_name, the_number, the_mgr, the_start);
end //
delimiter ;
```

- NOTE: Just like any other insert or update statement, they *must* abide by the rules of the table (e.g. primary key)
- Here, if we wanted to limit the number of departments an employee manages with this function, we could do the following…

```
drop procedure if exists add_new_dept_limited;
delimiter //
create procedure add_new_dept_limited(in the_name char(20),
    in the_number decimal(1,0), in the_mgr decimal(9,0), in the_start date)
sp_main: begin
    if (select count(*) from department where mgrssn = the_mgr) >= 2
        then leave sp_main; end if;

    insert into department values (the_name, the_number, the_mgr, the_start);
end //
delimiter ;
```

- You can even create tables with procedures…

```
drop procedure if exists dept_salary_filter_persistent;
delimiter //
create procedure dept_salary_filter_persistent(in low_bound integer,
    in high_bound integer, in the_dept integer)
begin
    drop table if exists dept_salary_filter_persistent_result;
    create table dept_salary_filter_persistent_result(
        fname char(10),
        lname char(20),
        ssn decimal(9,0),
        bdate date,
        address char(30),
        sex char(1),
        salary decimal(5,0),
        superssn decimal(9,0),
        dno decimal(1,0)
    );

    insert into dept_salary_filter_persistent_result
    select * from employee
    where salary between low_bound and high_bound
    and dno = the_dept;
end //
delimiter ;
```

Module 19: Database Design Guidelines

- Relational database design can be thought of as the grouping of attributes to form "good" relation schemas
- There are two levels of relational schemas…
    - *User (Logical) Level*
    - *Base (Storage) Level*
- Design is primarily concerned with the base level
- There are many criterion for creating "good" base relations; let's discuss some of them
- 1. Semantics of the Relational Attributes must be clear
    - Informally, each tuple in a relation should represent 1 entity or relationship instance
        - Attributes of different entities should *not* be mixed in the same relation
        - Only foreign keys should be used to refer to other entities
- 2. Design a Schema that does not suffer from insert, delete, or update anomalies
    - Update anomaly: Updating a relation causes an update on every relation using that relation
        - Ex. Changing the title of project P1 from "Billing" to "Customer-Accounting" causes an update to be made for all 100 employees
    - Insert anomaly
        - Ex. Cannot insert a project unless an employee is assigned to it. Conversely, cannot insert an employee unless they are assigned to a project
    - Delete anomaly
        - Ex. When a project is deleted, it will result in deleting all the employees who work on that project
        - Alternative, if an employee is the sole employee on that project, deleting that employee would result in deleting the corresponding project
- 3. Relations should be designed to have tuples with as few NULL values as possible
    - Attributes that rare NULL frequently could be placed in separate relations
    - Reasons for nulls include…
        - Attribute not applicable or invalid
        - Attribute value unknown
        - Value known to exist but unavailable
- 4. Relations should be designed to satisfy the "lossless join" condition
    - Bad design in relational databases may result in erroneous results for certain JOIN operations; the "lossless join" property is used to guarantee meaningful results for join operations
    - No spurious tuples should be generated by doing a natural-join of any relations

Module 20: Functional Dependencies

- DEFINITION: A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y. In other words, a **functional dependency** (**FD**) exists between X and Y.
- $X \rightarrow Y$ holds if whenever two tuples have the same value for X, they *must* have the same value for Y
    - For any two tuples t1 and t2 in any relation instance r(R): If $t1[X] = t2[X]$, then $t1[Y] = t2[Y]$
- Example

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

    - There's a functional dependency between Dnumber and Dname because every time Dnumber is 5, we can know for certain that Dname will be Research
    - Other examples include…
        - Social security number determines employee name (SSN → ENAME)
        - Project number determines project name and location (PNUMBER → {PNAME, PLOCATION})
- An FD is a property of the attributes in the schema R; the constraint must hold on *every* relation instance
- Demonstrating that a functional dependency exists is not necessarily a valid proof; in this case, to show that a functional dependency between attributes, we should rather show what attributes don't have a functional dependency
- Example
    - Which FDs *may* exist in this relation?

A	B	C	D
a1	b1	c1	d1
a1	b2	c2	d2
a2	b2	c2	d3
a3	b3	c4	d3

    - A does not have an FD between B, C, or D.
    - B does not have an FD with D. It may have an FD with C.
    - C does not have an FD with D.
- NOTE: If K is a key of R, then K functionally determines *all* attributes in R since we never have two distinct tuples with the same key
- Given the instance of a relation, all we can conclude is that an FD *may exist* between certain attributes; we can only definitely conclude that FDs *do not exist*

Module 21: Normalization & 1NF, 2NF, and 3NF

Normalization of Relations

- DEFINITION: **Normalization** is the process of decomposing unsatisfactory or "bad" relations by breaking up their attributes into smaller relations
- DEFINITION: **Normal form** is a condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form
- There are many different tiers of normalization…
    - 2NF, 3NF, BCNF: based on keys and FDs of a relation schema
    - 4NF: based on keys, multivalued dependencies (MVDs)
    - 5NF: based on keys, join dependencies (JDs)
- NOTE: Just because something is in normal form does not ensure we have good relational design. We may need to also consider lossless joins and other properties of the sort.
- The practical utility of these normal forms become questionable when the constraints on which they are based are hard to understand or detect in the first place
- NOTE: Just because we *can* normalize at the highest possible form, doesn't always mean we should. In fact, designers usually only go up to 3NF and BCNF; 4NF is rarely used in practice
- DEFINITION: **Denormalization** is the process of storing the join of higher normal form relations as a base relation, which is in a lower normal form

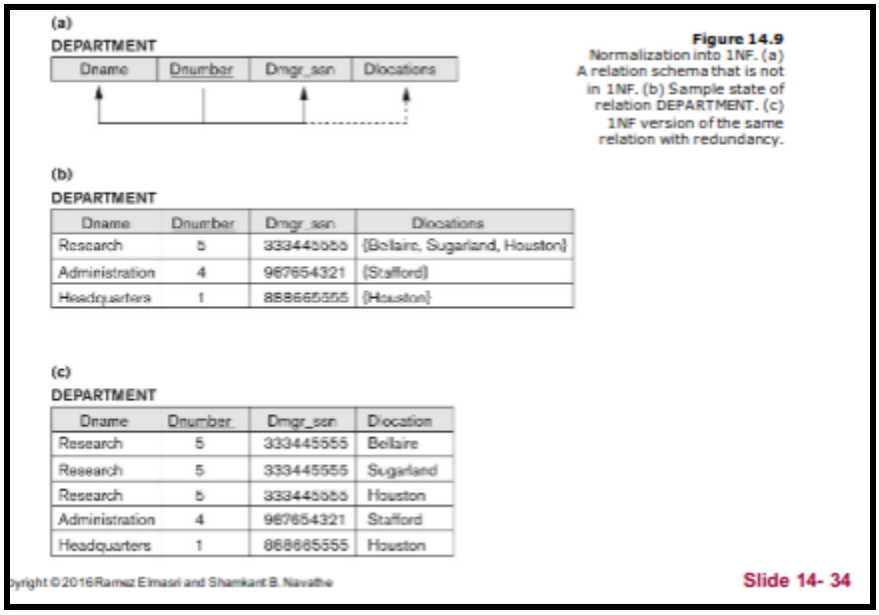
Keys & Superkeys

- DEFINITION: A **superkey** of a relation schema $R = \{A_1, A_2, \ldots, A_n\}$ is a set of attributes $S \subset R$ with the property that no two tuples t1 and t2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
    - With this definition, then we can reframe our definition of a key: a *key* K is a superkey with the additional property that the removal of any attribute from K will cause K not to be a superkey anymore
- DEFINITION: If a relation schema has more than one key, each is called a **candidate key**
    - One of these is arbitrarily designated as the primary keys, while others are called secondary keys
- DEFINITION: A **prime attribute** must be a member of *some* candidate key; a **nonprime attribute** is not a member of any candidate key

First Normal Form

- In first normal form, relations are not allowed to have…
    - Composite attributes

- Multivalued attributes
- Nested relations (attributes whose values for an individual tuple are non-atomic)



(a)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations

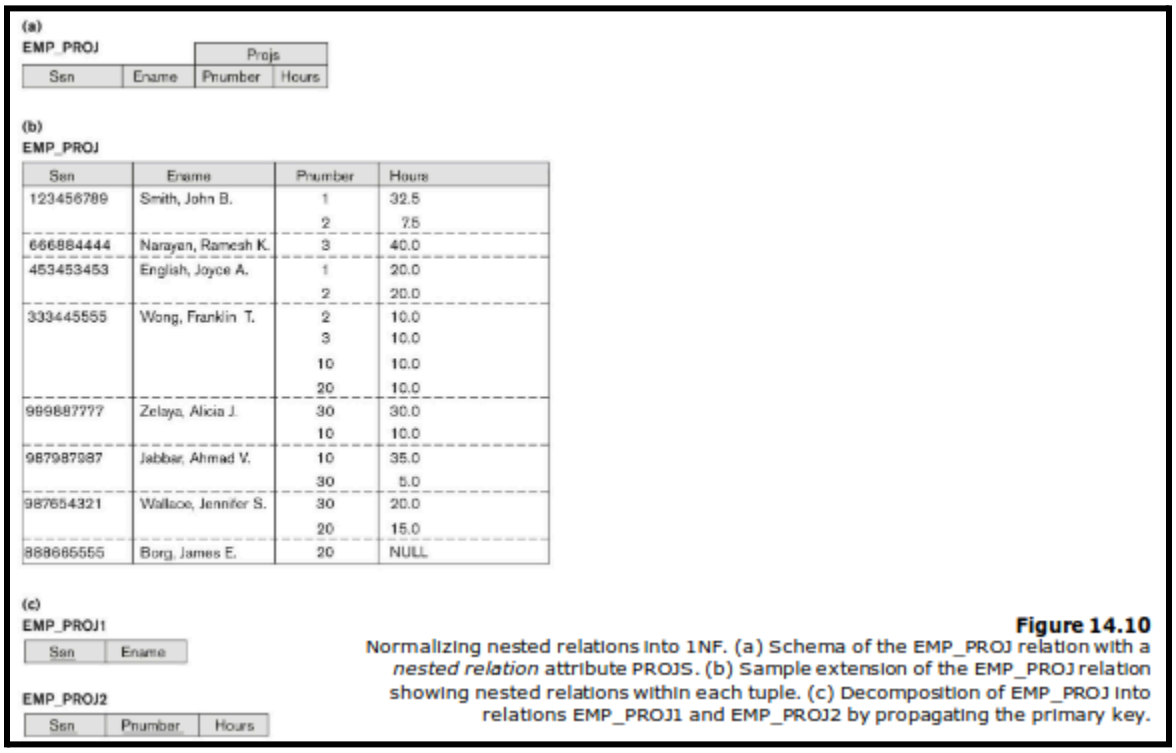
(b)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 14.9
Normalization into 1NF. (a) A relation schema that is not in 1NF. (b) Sample state of relation DEPARTMENT. (c) 1NF version of the same relation with redundancy.



(a)
EMP_PROJ

Ssn	Ename	Projs	
		Pnumber	Hours

(b)
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)
EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Figure 14.10
Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a *nested relation* attribute PROJS. (b) Sample extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposition of EMP_PROJ into relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

Second Normal Form

- Compared to 1NF, 2NF uses FDs and the concept of primary keys
- DEFINITION: A **prime attribute** is an attribute that is a member of the primary key K
- DEFINITION: A **full functional dependency** is an FD Y→Z where removal of any attribute from Y means the FD does not hold any more

    - In other words, every attribute of Y has a functional dependency with every attribute of Z
- Examples
    - {SSN, PNUMBER} →HOURS is a full FD since neither SSN → HOURS nor PNUMBER → HOURS hold
    - {SSN, PNUMBER} → ENAME is not a full FD since SSN → ENAME also holds
- DEFINITION: A relation schema R is in **second normal form** (**2NF**) if every non-prime attribute A in R is fully functionally dependent on the primary key
- R can be decomposed into 2NF relations via the process of 2NF normalization

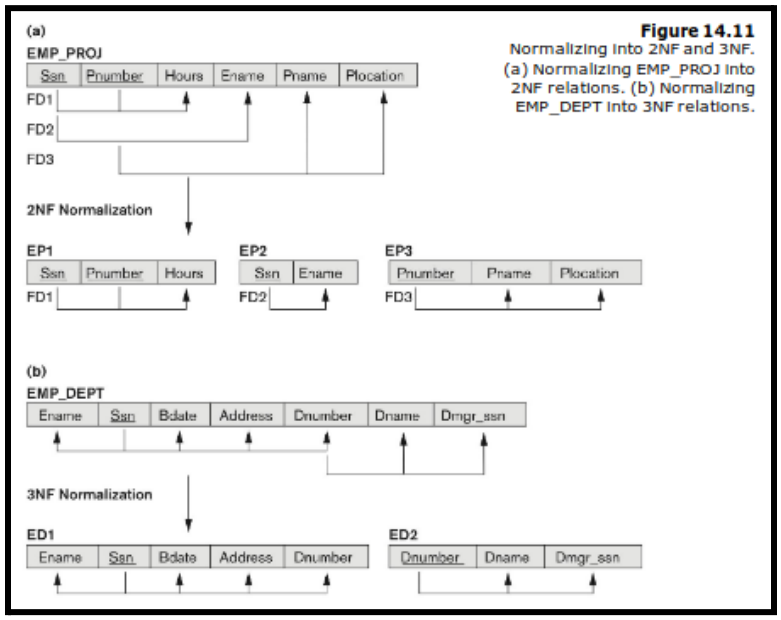


Figure 14.11
Normalizing into 2NF and 3NF. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.

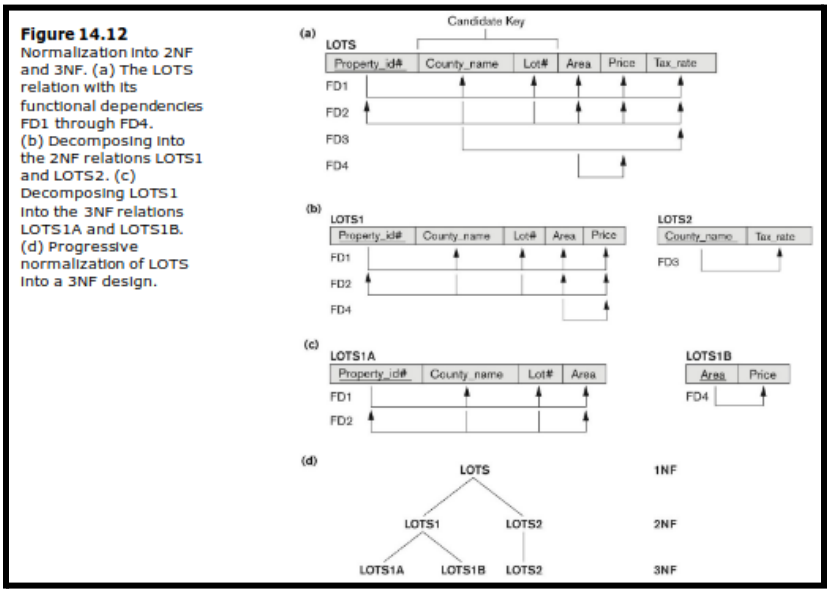


Figure 14.12
Normalization into 2NF and 3NF. (a) The LOTS relation with its functional dependencies FD1 through FD4. (b) Decomposing into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Progressive normalization of LOTS into a 3NF design.

Third Normal Form

- DEFINITION: A **transitive functional dependency** as an FD X → Z that can be derived from two FDs X →Y and Y →Z
- Examples
    - SSN →DMGRSSN is a transitive FD since SSN → DNUMBER and DNUMBER → DMGRSSN hold
    - SSN →ENAME is not transitive since there is not set of attributes X where SSN → X and X → ENAME
- DEFINITION: A relation schema R is in **third normal form** (**3NF**) if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
- Similar to the process of 2NF normalization, we can reach 3NF through 3NF normalization
- There is one stipulation about 3NF: if our attribute Y in X →Y and Y → Z is a candidate key, we say there is *no problem* with the transitive dependency
    - This sort of makes sense; since all the candidate keys would be unique for each tuple, then of course you could insert a candidate key in the middle of the FD to make it transitive. In fact, if every value in the tuple is a candidate key then one could see how we could include every attribute in a string of FDs

Informal Overview of the Normalizations

- In 1NF, all attributes depend on the *key*
- In 2NF, all attributes depend on the *whole key*
- In 3NF, all attributes depend on *nothing but the key*

Module 22: Relational Algebra - Basic Operations

- DEFINITION: **Relational algebra** is the most basic set of operations for the relational model; they enable a user to specify basic retrieval requests (sounds a lot like querying, right?)
    - A sequence of relational algebra operations forms a *relational algebra expression*
- The result of an operation is a new *relation*, which may have been formed from one or more input relations
    - Thus, using these algebra operations produce new relations

Overview of Operations

- Unary Relational Operations
    - SELECT (symbol: σ (sigma))
    - PROJECT (symbol: π (pi))
    - RENAME (symbol: ρ (rho))
- Relational Algebra Operations From Set Theory
    - UNION (∪), INTERSECTION (∩), DIFFERENCE (or MINUS, –)
    - CARTESIAN PRODUCT (x)
- Binary Relational Operations
    - JOIN (several variations of JOIN exist)
    - DIVISION
- Additional Relational Operations
    - OUTER JOINS, OUTER UNION
    - AGGREGATE FUNCTIONS (These compute summary of information: for example, SUM, COUNT, AVG, MIN, MAX)

- We may want to apply several relational algebra operations together by either making one *relational algebra expression* or we can do one operation at a time and create intermediate result relations
- Ex.
    - $\pi_{FNAME, LNAME, SALARY}(\sigma_{DNO=5}(EMPLOYEE))$ is a single algebraic expression, but it can be rewritten as a sequence of operations…

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$

$$RESULT \leftarrow \pi_{FNAME, LNAME, SALARY}(DEP5\_EMPS)$$

- SELECT
    - The SELECT operation (denoted by σ (sigma)) is used to select a subset of the tuples from a relation based on a selection condition
        - This selection condition acts as a *filter*

        - Tuples satisfying the condition are selected; others are discarded (filtered out)
    - Ex. Select the EMPLOYEE tuples whose department number is 4

$$\sigma_{DNO = 4}(EMPLOYEE)$$

    - In general, the select operation is denoted by…

$$\sigma_{<selection\ condition>}(R), \text{ where}$$

        - Sigma is the select operator
        - The selection condition is a boolean expression specified on the attributes of relation R
        - Tuples that make the condition TRUE are select, tuples that make the condition FALSE are discarded
    - The SELECT operation has the following properties…
        - It produces a relation S that has the same schema (same attributes) as R
        - SELECT is commutative

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

            - Because of this, it means select can be done in any sequence to get the same result
            - A cascade of SELECTs can be replaced by one single selection with a conjunction on the conditions

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond1>\ AND\ <cond2>}(R)$$

        - The number of tuples in the result of a SELECT is ≤ the number of tuples in the input relation R

- <u>PROJECT</u>
    - The PROJECT operation, denoted by π (pi), keeps certain columns (attributes) from a relation and discards the other columns
        - In other words, it creates a vertical partitioning
        - Any attribute not selected in the tuple is discarded
    - Ex. Selecting each employee's first and last name and salary

$$\pi_{LNAME, FNAME, SALARY}(EMPLOYEE)$$

    - The general form of the PROJECT operation is…

$$\pi_{<attribute\ list>}(R), \text{ where}$$

        - Pi is the PROJECT operation
        - <attribute list> is the desired attributes from the relation R
    - NOTE: The PROJECT operation removes any duplicate tuples. This is because the result of the project operation must be a *set* of tuples, and mathematical sets do not allow duplicate elements
    - The PROJECT operation has the following properties…

        - The number of tuples in the resulting projection is always ≤ to the number of tuples in R
        - PROJECT is *not* commutative
- RENAME
    - The RENAME operator, denoted by ρ (rho), allows us to rename the attributes of a relation or the relation name or both
    - May be necessary in the case of certain JOIN operations
    - The general RENAME operation ρ can be expressed by any of the following forms
        - $\rho_{S(B1, B2, \ldots, Bn)}(R)$ changes both…
            - The relation name to S, and
            - The column (attribute) names to B1, B2, …, B_n
        - $\rho_{S}(R)$ changes the relation name only to S
        - $\rho_{(B1, B2, \ldots, Bn)}(R)$ changes the column attribute names to B1, B2, …, B_n
- UNION
    - The UNION operator, denoted by ∪, creates a relation from relation S and relation R in which the new relation includes all tuples that are either in R or in S or in both R and S
    - Duplicate tuples are eliminated (analogous to the union operator in MySQL)
    - The two operand relations R and S must be "type compatible" (or UNION compatible)
        - R and S must have the same number of attributes
        - Each pair of corresponding attributes must be type compatible
        - NOTE: These same restrictions are in place for INTERSECTION and SET DIFFERENCE as well
    - Example

- To retrieve the social security numbers of all employees who either *work in department 5* (RESULT1 below) or *directly supervise an employee who works in department 5* (RESULT2 below)
- We can use the UNION operation as follows:

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{SSN}(DEP5\_EMPS)$$
$$RESULT2(SSN) \leftarrow \pi_{SUPERSSN}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

- INTERSECTION
  - The INTERSECTION operator, denoted by ∩, is done on two relations R and S and creates a new relation that includes all the tuples in both R and S
    - The attribute names in the result will be the same as the attribute names in R
  - The two operand relations R and S must be "type compatible"
- SET DIFFERENCE
  - The SET DIFFERENCE operator (or MINUS or EXCEPT), denoted by -, uses two relations R and S and creates a resulting relation R - S which includes all tuples that are in R but not in S
    - The attribute names in the result will be the same as the attribute names in R
  - The two operand relations R and S must be "type compatible"
- Properties of UNION, INTERSECT, and DIFFERENCE
  - Union and intersection are *commutative* operations…

$$R \cup S = S \cup R, \& R \cap S = S \cap R$$

  - Union and intersection are *associative* operations…

$$R \cup (S \cup T) = (R \cup S) \cup T$$

$$(R \cap S) \cap T = R \cap (S \cap T)$$

  - The minus operation is *not* commutative; that is, in general

$$R - S \neq S - R$$

- CARTESIAN PRODUCT
  - The CARTESIAN (or CROSS) PRODUCT operation, denoted by $R(A_1, A_2 \ldots, A_n)$ x $S(B_1, B_2, \ldots, B_m)$ creates a relation Q with n + m attributes $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots, B_m)$
  - The resulting relation state has one tuple for each combination of tuples--one from R and one from S

- Hence, is R has n_R tuples and S has n_S, then R x S will have n_R * n_S tuples
- The two operands do NOT have to be "type compatible"
- Generally, CROSS PRODUCT is not a meaningful operation, but it can become meaningful when followed by other operations
- Example…

- Example (not meaningful):
  - FEMALE_EMPS ← $\sigma_{SEX='F'}$(EMPLOYEE)
  - EMPNAMES ← $\pi_{FNAME, LNAME, SSN}$ (FEMALE_EMPS)
  - EMP_DEPENDENTS ← EMPNAMES x DEPENDENT
- EMP_DEPENDENTS will contain every combination of EMPNAMES and DEPENDENT
  - whether or not they are actually related

- To keep only combinations where the DEPENDENT is related to the EMPLOYEE, we add a SELECT operation as follows
- Example (meaningful):
  - FEMALE_EMPS ← $\sigma_{SEX='F'}$(EMPLOYEE)
  - EMPNAMES ← $\pi_{FNAME, LNAME, SSN}$ (FEMALE_EMPS)
  - EMP_DEPENDENTS ← EMPNAMES x DEPENDENT
  - ACTUAL_DEPS ← $\sigma_{SSN=ESSN}$(EMP_DEPENDENTS)
  - RESULT ← $\pi_{FNAME, LNAME, DEPENDENT\_NAME}$ (ACTUAL_DEPS)
- RESULT will now contain the name of female employees and their dependents

- <u>JOIN</u>
  - The JOIN operation, denoted by ⋈, emulates a sequence of CARTESIAN PRODUCT operations followed by a SELECT
  - The general form of the join operation on two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is:

$$R \bowtie_{<join\ condition>} S$$

  - Some properties of JOIN…
    - Result is a relation Q with degree $n + m$ attributes

        - The resulting relation state has one tuple for each combination of tuples only if they satisfy the join condition
        - Only related tuples based on the join condition will appear in the result
    - The general case of JOIN is called a theta-join…

$$R \bowtie_{theta} S$$

    - Theta can be any general boolean expression on the attributes of R and S, for example: $R.A_i < S.B_j$
- <u>EQUIJOIN</u>
    - The most common use of join involves join conditions with equality comparisons only
    - Such a join, where the only comparison operator used is =, is called an EQUIJOIN
- <u>NATURAL JOIN</u>
    - Another variation of JOIN called NATURAL JOIN, denoted by *, was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition
        - Because one of each pair of attributes with identical values is superfluous
    - The standard definition of natural join requires that the two join attributes, or each pair of corresponding join attributes have the same name in both relations

Aggregate Functions and Grouping as Relational Operations

- A type of request that cannot be expressed in the basic relational algebra is to specify mathematical aggregate functions on collections of values from a database
- To do this, we use the Aggregate Functional operation F
    - $F_{MAX\ Salary}(EMPLOYEE)$ retrieves the maximum salary value from the EMPLOYEE relation
    - $F_{MIN\ Salary}(EMPLOYEE)$ retrieves the minimum salary value from the EMPLOYEE relation
    - $F_{SUM\ Salary}(EMPLOYEE)$ retrieves the sum of the salary from the EMPLOYEE relation
    - $F_{COUNT\ SSN,\ AVERAGE\ Salary}(EMPLOYEE)$ computes the count of employees and their average salary
- We can combine grouping with Aggregate Functions
    - Ex. For each department, retrieve the DNO, COUNT SSN, and AVERAGE SALARY
    - A variation of aggregate operation F allows this…
        - Grouping attribute placed to left of symbol
        - Aggregation functions to right of symbol

$$_{DNO}F_{COUNT\ SSN,\ AVERAGE\ Salary}(EMPLOYEE)$$