CS 157A
Introduction to Database Management Systems

Lecturer: Tahereh Arabghalizi, PhD (She, Her)

Department of Computer Science

SJSU SAN JOSÉ STATE UNIVERSITY

Important Reminder

- Don’t rely on the review slides!
- Make sure to **study all the lecture slides**
- The review is just a summary, but the details and concepts we covered in class are just as important!

Introduction to Databases

DBMS

- A Database Management System (DBMS) is a <u>software package </u>designed to **store** and **manage** and **retrieve** data efficiently.

- A DBMS not only stores data but also ensures data **integrity**, **security**, and **efficient access**.

DBMS Advantages

- **Data independence and efficient access:**
  - Applications are insulated from how data is structured and stored.
- **Data integrity and security:**
  - Ensuring that the data is accurate and accessible only to authorized users.
- **Concurrent access, high availability:**
  - Multiple users can access the data simultaneously without issues.
- **Transaction management, Recovery from crash:**
  - Ensures data integrity even in cases of system failures.

What is a RDBMS System?

A Relational Database Management System (RDBMS) is a type of DBMS that **stores data** in a **structured format** using rows and columns, much like a spreadsheet.

This tabular format:

- Manages **very large** amounts of data
- Supports **efficient access** to very large amounts of data
- Supports **concurrent access** to very large amounts of data
- Supports **secure access** to very large amount of data
- Supports **atomic access** to very large amount of data (maintains the integrity and reliability of the database)

High-level Overview of RDBMS (1)

- **Data Definition Language (DDL):** used to **define** the database schema, such as creating tables and indexes.

- **Data Manipulation Language (DML):** used to **manipulate** existing tables (Insert, Update, Delete)

- **SQL Processing**: user typically interact with RDBMS through either a query or a DML statements to manipulate the existing content of the database. The query processor **translates** SQL queries into executable **actions**.

High-level Overview of RDBMS (2)

A query processor consists of the following <u>two</u> main components:

- **SQL Compiler:** translates SQL statement into internal representation called a "query plan." The query plan is a sequence of actions that will be executed by the execution engine:
  - <u>Query parser</u>: builds a **tree structure** from the SQL text
  - <u>Query preprocessor</u>: performs **semantic checking** like relations accessed by the query actually exists, and transform the parse tree into **tree of algebraic operators** representing **the query plan**
  - <u>Query optimizer</u>: optimizes the query plan for efficient execution.
  - **Execution engine:** executes the sequence of operations in the query plan.

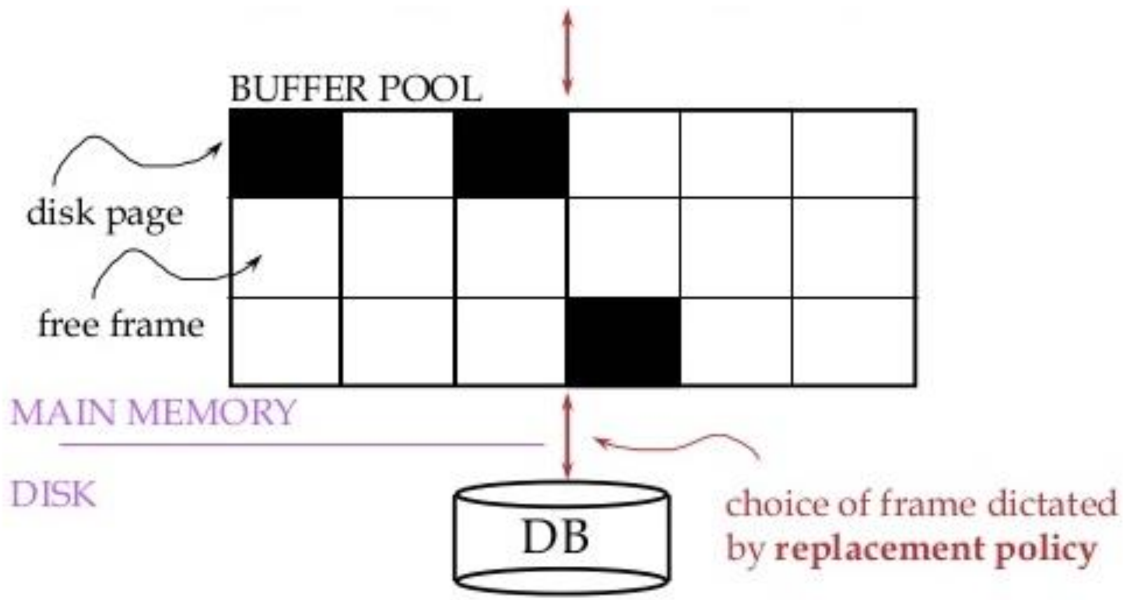
High-level Overview of RDBMS (3)

- **Transaction Processing**: queries and DML statements are grouped into transactions to provide ACID (Atomicity, Consistency, Isolation, Durability) properties:
  - **Atomicity** ensures that a transaction is treated as a single, indivisible unit of work. (All or nothing!)
  - **Consistency** ensures that a transaction takes the database from one valid state to another valid state, maintaining all predefined rules.
  - **Isolation** ensures that transactions are executed independently of one another.
  - **Durability** ensures that once a transaction has been committed, its results are permanent, even in the case of a system failure.

Example: withdrawing money from an ATM

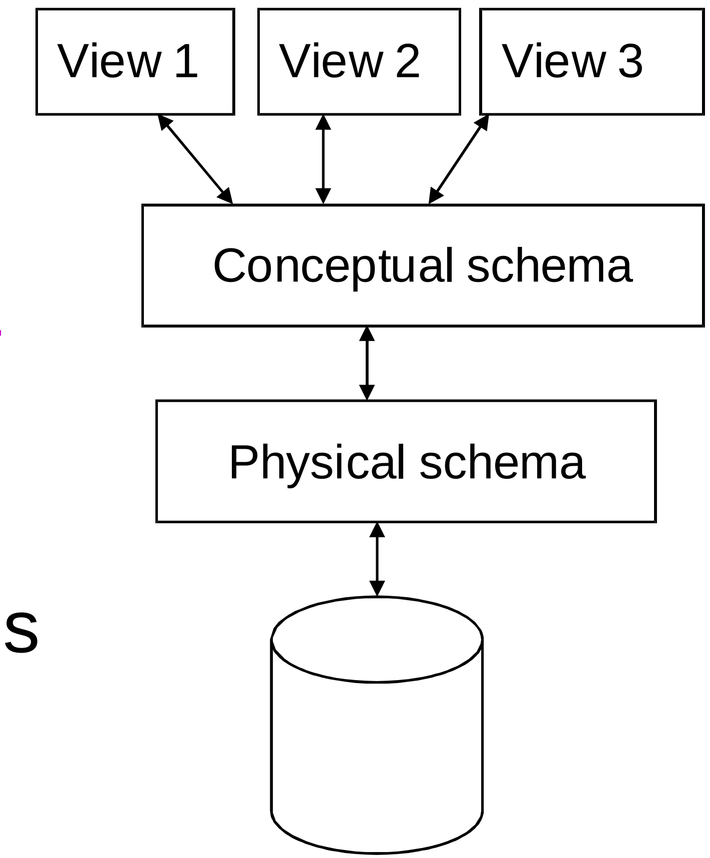
High-level Overview of RDBMS (4)

- **Storage and Buffer Management**: Data in the RDBMS reside on secondary storage (disk). To do anything useful work, we need to bring the data into memory (buffer cache).
- **Storage manager** controls the placement of data on disk and movement between disk and main memory.



Levels of Abstraction

- **Many views**
  - Describes how users see the data
- **Single conceptual (logical) schema**
  - Define logical structure of the data
- **Single physical schema**
  - Describes how data is stored in files and indexes (e.g., B-tree).



Intuitive example: designing a house

Data Models

What is a Data Model?

1. **Mathematical representation of data describing how data is structured and manipulated:**
   - **Examples:** relational model => tables
     semi-structured model => trees/graphs, XML, etc.

2. Defines **operations** on data
3. Defines **constraints** that must be maintained

Schemas

A schema is a **blueprint** of a database.

- *Relation schema* = relation name and attribute list.
  - **Optionally**: types of attributes.
  - **Example**: Beers(name, manf) or
    Beers(name: string, manf: string)

- *Database schema* = set of all relation
  schemas in the database.

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
        <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Elements of Table Declarations

- Most basic element: an attribute and its type.

- The most common types are:
  - **INT** or INTEGER (synonyms).
  - **REAL** or FLOAT (synonyms).
  - **CHAR(*n*)** = fixed-length string of *n* characters.
  - **VARCHAR(*n*)** = variable-length string of up to *n* characters.

Example: Create Table (Contd.)

- **Modifying Relation schemas:**
  - ALTER TABLE <name> ADD phone CHAR(16);
  - ALTER TABLE <name> DROP phone;

- **Default values:**
  - Title CHAR(100) DEFAULT “UNKNOWN”
  - ALTER TABLE <name> ADD phone CHAR(16) DEFAULT ‘unlisted’

- **Declaring keys:**
  - Key is an attribute or list of attributes
  - **Key types:** PRIMARY KEY or UNIQUE
  - The above keys says no two tuples of a relation will have same key

Semi-structured Data

- Another data model, based on trees.

- Motivation: flexible representation of data.
  - often used when the structure of the data can change over time or is not strictly defined.

- Motivation: sharing of *documents* among systems and databases.

Graphs of Semi-structured Data

- Nodes = objects.
- Labels on arcs (like attribute names).
- Atomic values are at leaf nodes (nodes with no arcs out).
- Flexibility - no restrictions on:
  - Labels out of a node.
    - **Example:** A node representing a **person** could have labels like **Person**, **Employee**, and **Customer** simultaneously.
  - Number of successors with a given label.
    - **Example:** A **Person** node can have multiple **FRIEND** relationships pointing to many other **Person** nodes without limitations.

XML

- XML = *Extensible Markup Language*.
- While HTML uses tags for **formatting** (e.g., "italic"), XML uses tags for **semantics** (e.g., "this is an address").
- Key idea: create **custom tag** sets for specific domains (e.g., genomics), and translate all data into properly tagged XML documents.
- This flexibility makes XML a powerful tool for data interchange and storage.

DTD's (Document Type Definitions)

- A grammatical notation for describing **allowed use of tags**.
- Definition form:

```
<!DOCTYPE <root tag> [
  <!ELEMENT <name>(<components>)>
  . . . more elements . . .
]>
```

Relational Algebra

What is Relational Algebra?

- An *algebra* whose <u>operands</u> are **relations** or variables that represent relations.

- The <u>operators</u> are designed to **perform the most common operations** needed for managing relations in a database, forming a **query language** for relations.

Core Relational Algebra

- Union, intersection, and difference:
  - Usual set operations, but *both operands must have the same relation schema* *(Same Number of Columns, Same Data Types, and Same Order of Columns).*
- Selection: picking certain rows.
- Projection: picking certain columns.
- Products and joins: compositions of relations.
- Renaming of relations and attributes.

Selection

- R1 := σ_C (R2)
  - *C* is a condition (as in “if” statements) that refers to attributes of R2.
  - R1 is all those tuples of R2 that satisfy *C*.

Projection

- R1 := π_L (R2)
  - *L* is a list of attributes from the schema of R2.
  - R1 is constructed by looking at each tuple of R2, extracting the attributes on list *L*, **in the order specified**, and creating from those components a tuple for R1.
  - Eliminate duplicate tuples, if any.

Extended Projection

- Using the same π_L operator, we allow the list L to contain arbitrary expressions involving attributes: for example:
  1. Arithmetic on attributes, e.g., $A+B \rightarrow C$.
  2. Duplicate occurrences of the same attribute.

Product (Cartesian Product)

- R3 := R1 X R2
  - Pair each tuple t1 of R1 with each tuple t2 of R2.
  - Concatenation t1t2 is a tuple of R3.
  - Schema of R3 is the attributes of R1 and then R2, in order.
  - **But** be aware an attribute *A* of the same name in R1 and R2: use R1.*A* and R2.*A*

Theta-Join

- $R3 := R1 \bowtie_C R2$
  - Take the product R1 X R2.
  - Then apply σ_C to the result.
- *C* can be any Boolean-valued condition.

Natural Join

- A useful join variant connects two relations by:
  - Equating attributes of the same name, and
  - Projecting out duplicate columns.
- Denoted $R3 := R1 \bowtie R2$.

Renaming

- The ρ operator gives a new schema to a relation.
- $R1 := \rho_{R1(A1,\ldots,An)}(R2)$ makes R1 be a relation with **attributes** A1,…,A*n* and the **same tuples** as R2.
- Simplified notation: R1(A1,…,A*n*) := R2.

Building Complex Expressions

- Combine operators with **parentheses** and **precedence rules**.
- Three notations, just as in arithmetic:
  1. Sequences of assignment statements.
  2. Expressions with several operators.
  3. Expression trees.

Constraints on Relations

- **Relational Algebra as a Constraint Language**
  - Two ways to use expressions of relational algebra to express constraints:
    - If R is an expression, then **R = 0** is a constraint – no tuples in the result R
    - If R and S are expressions of relational algebra, then $\mathbf{R \subseteq S}$ is a constraint that says **every tuple in R** must also be in S. Of course S may contain additional tuples not in R

      Example: $\pi_{CustomerID}(Orders) \subseteq \pi_{CustomerID}(Customers)$

- **Referential Integrity Constraints**
  - Referential Integrity constraint asserts that <u>a value </u>appearing in one context (**typically in a foreign key column**) also appears in <u>another related context </u>(**usually in a primary key column of another table**).

Constraints on Relations (Contd.)

- **Key Constraints**
  - To express algebraically that an attribute or set of attributes is a key for a relation R
  - Let us use hypothetical two names for the same relation: R and S and the attribute key is name, and another random attribute is address:
  - $\sigma_{\text{R.name=S.name AND R.address} \neq \text{S.address}}$ **(R x S) = 0**

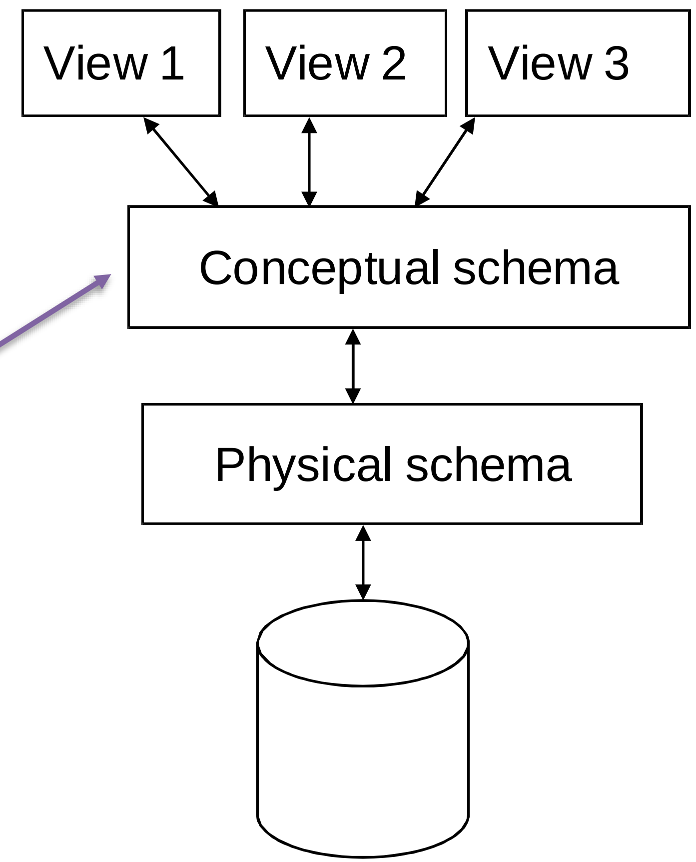
- **Additional Constraints**
  - Assume gender of movie star has to be ‘M’ or ‘F’ only
  - $\sigma_{\text{gender} \neq \text{'M' AND gender} \neq \text{'F'}}$ **(MovieStar) = 0**

Why Bags?

- SQL, the most important query language for relational databases, is actually a **bag language**.

- Some operations, like projection, are more **efficient** to produce bags than sets:
  - No need for duplicate elimination
  - Reduced Computational Complexity
  - Simpler Implementation

Logical Database Model: Entity Relationship Model



Purpose of E/R Model

- The E/R model allows us to **sketch** database schema designs:
  - Includes some constraints, <u>but not operations</u>!
- Designs are pictures called *entity-relationship (E/R) diagrams*
- Later: convert E/R designs to relational DB designs (i.e., schema)

Key Concepts

- ***Entity*** = “thing” or object (tuple)
- ***Entity set*** = collection of similar entities; typically implemented as a relation (relation)
  - Similar to a class in object-oriented languages.
- ***Attribute*** = property of (the entities of) an entity set:
  - Attributes are simple values, e.g. integers or character strings, etc.
- **Relationships** = connection between two or more entity sets – binary or multi-way relationships.

E/R Diagrams

- **In an entity-relationship diagram:**
  - Entity set = rectangle
  - Attribute = oval, with a line to the rectangle representing its entity set
  - Relationship = diamond, connecting two or more entity sets

Many-One & One-Many Relationships

- In a **many-one** (E-F) relationship R connecting entity sets E and F: each entity in E is connected to zero or one entity in F. Member in F can be connected to 0, 1, or many entities in E.

- In a **one-many** (E-F) relationship R connecting entity sets E and F: an entity in E is connected to 0, 1, or many entities in F. Member in F is connected to zero or one entity in E.

- In **one- one** relationship an entity in E or F can be connected to **at most one entity (i.e., 0 or 1)** of the other set.

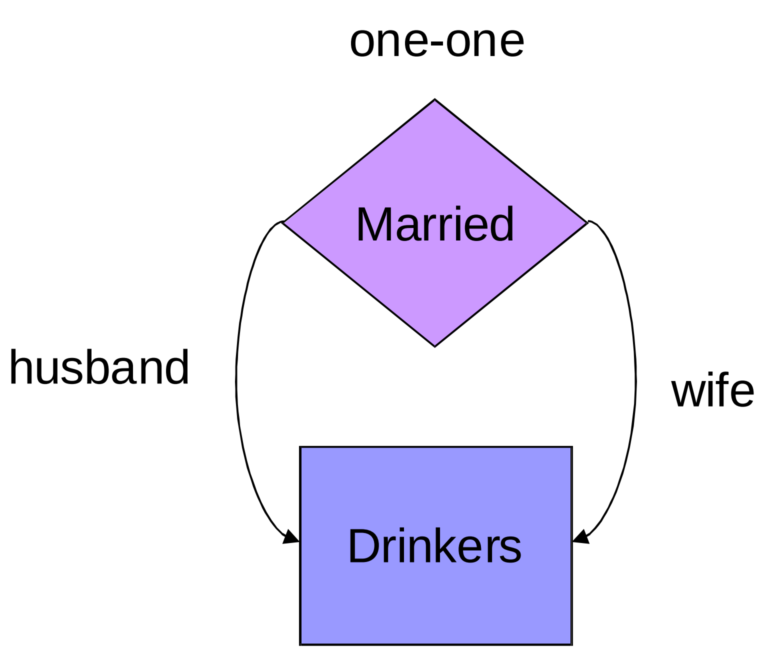
- In a ***many-many* relationship**, an entity of either set can be connected to many entities of the other set.

Representing "Multiplicity"

- Show a **many-one** relationship by an arrow entering the **"one" side**
- Show a **one-one** relationship by arrows entering **both** entity sets – an arrow means "at most one"
- Rounded arrow equal "**exactly one**," i.e., each entity of the first set is related to exactly one entity of the target set.

Roles

- Sometimes an entity set appears **more than once** in a relationship.
- Label the edges between the relationship and the entity set with names called *roles.*

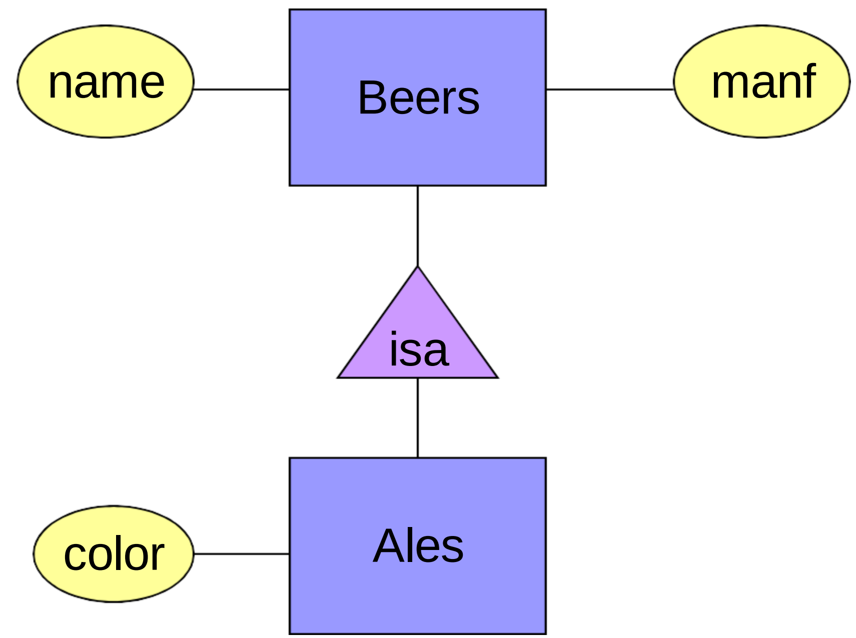


Relationship Set

Husband	Wife
Bob	Ann
Joe	Sue
…	…

Subclasses in E/R Diagrams

- Assume subclasses form a **tree**
  - i.e., No multiple inheritance
- isa **triangles** indicate the subclass relationship of type one-one relationship without arrows
  - Point to the superclass

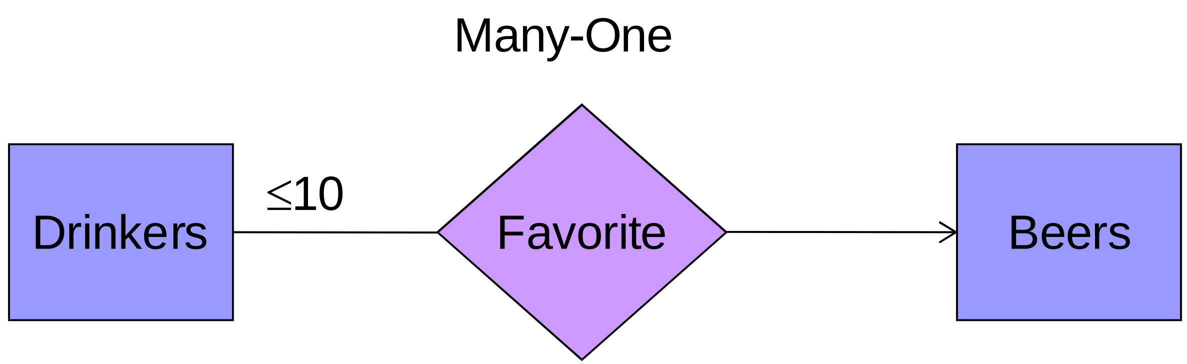


Keys in E/R Diagrams

- <u>Underline</u> the key attribute(s)

- In an **isa** hierarchy, **only the root** entity set has a key (a constraint), and it must serve as the <u>key for all entities in the hierarchy</u>.

Degree Constraints



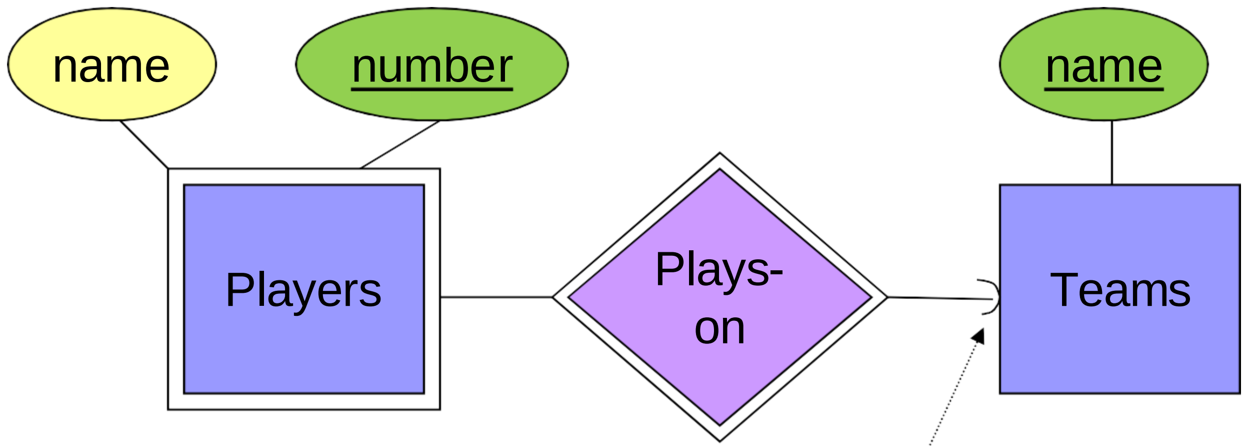
- We can attach a **bounding number** to the edge connecting a relationship to an entity set
- The attached number indicates **limits on the number of entities** that can be connected to any single entity of the related entity set
- In the above example: only ≤10 Drinkers can have the same Favorite Beer

Weak Entity Sets

- Occasionally, entities of an entity set need “**help**” to identify them <u>uniquely.</u>

- **Weak entity** set is an entity set with its key’s attributes, **some or all** belong to other entity sets

In E/R Diagrams



Note: "Players" is a week entity set because player name & number are not unique unless associated with a team

Note: must be rounded because a player can be a member of only one team

- To identify player uniquely, we need player number + team's name as the key!
- Double diamond for *supporting week* many-one relationship
- Double rectangle for the *weak* entity set

Design Techniques- Avoiding Redundancy

- *Redundancy* = saying the same thing in two (or more) different ways
- Wastes space and (more importantly) encourages inconsistency! Why?
  - Two representations of the same fact become inconsistent if we change one and forget to change the other.

Design Techniques - Entity Sets Versus Attributes

- An entity set should satisfy at least one of the following conditions, otherwise the entity set should be **merged** with another entity set and not to stand on its own:
  - it has at least one non-key attribute

    or

  - It is the "many" in a many-one or many-many relationship

Design Techniques- Don’t Overuse Weak Entity Sets

- Beginning database designers often doubt that anything could be a key by itself:
  - They make all entity sets weak, supported by all other entity sets to which they are linked
- In reality, we usually create **unique ID’s** for entity sets:
  - Examples include social-security numbers, automobile VIN’s etc.

From E/R Diagrams to Relations

- **Entity set** → relation.

  Entity → tuple or row

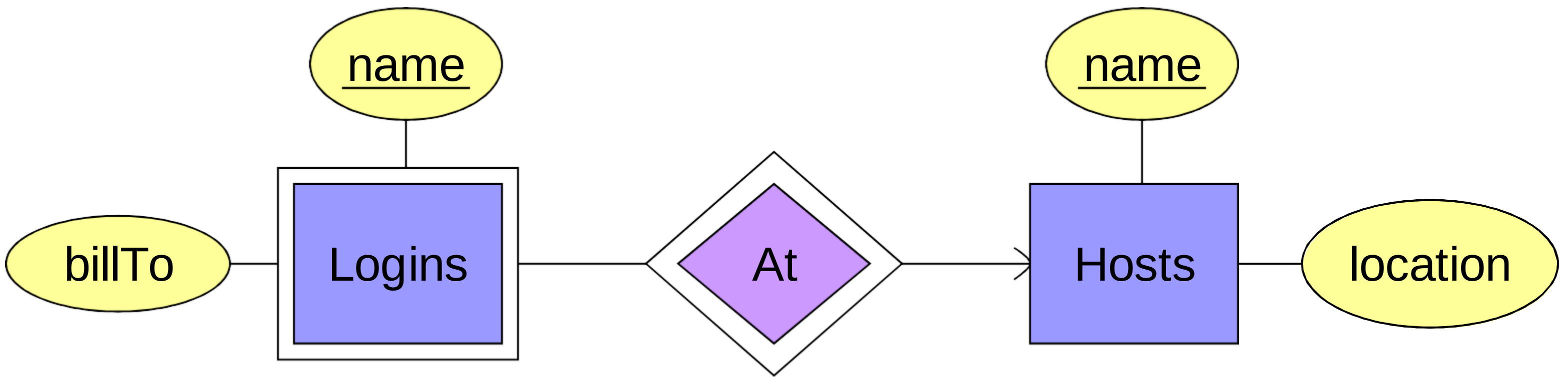
  Attributes → attributes.

- **Relationships** → relations, whose attributes are <u>only</u>:
  - The keys of the connected entity sets => foreign keys
  - Attributes of the relationship itself, if any.

Handling Weak Entity Sets

- Relation for a **weak entity set** must include attributes for its **complete key** (including those belonging to other entity sets), as well as its own, non-key attributes.
- A **supporting relationship** is redundant and yields no relation (unless it has attributes).

Example: Weak Entity Set → Relation



Hosts(<u>hostName</u>, location)

Logins(<u>loginName</u>, <u>hostName</u>, billTo)

~~At(loginName, hostName, hostName2)~~

Must be the same

At becomes part of Logins

Placeholder for Future Expansion

Subclasses (isa)

E/R style:

One relation for each entity set E (subclass) in the hierarchy:

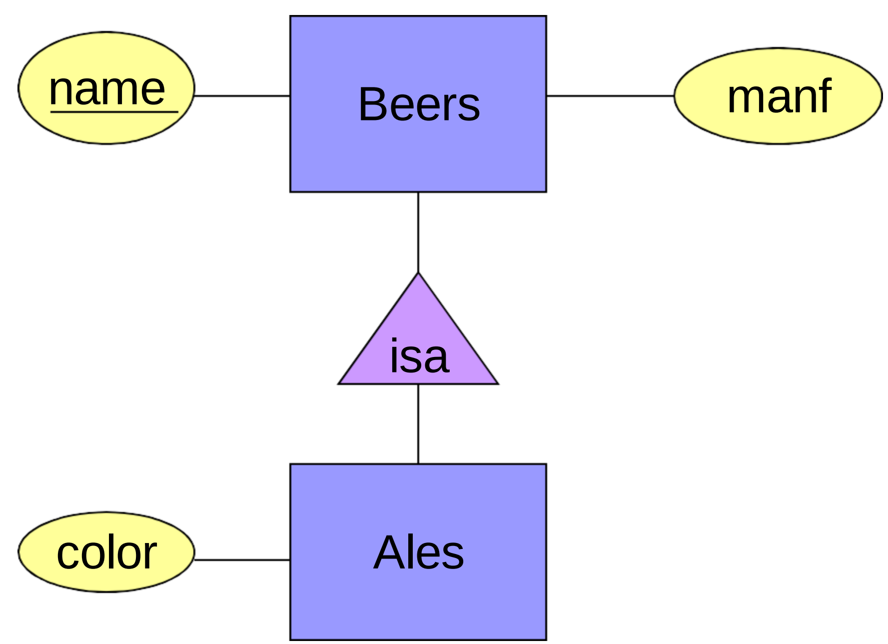
- Key attribute(s) from the root
- Attributes of that subclass

Beers

name	manf
Bud	Anheuser-Busch
Summerbrew	Pete's

Ales

name	color
Summerbrew	dark



Normalization

Normalization

- **Normalization** is the process of organizing data in a database to minimize ***redundancy*** and ***dependency***.

- The goal is to ensure that each piece of data is stored **only once** and that the database remains **efficient** and **consistent**.

- It involves **dividing** large tables into smaller, related tables and defining relationships between them.

- Normalization is carried out through a series of steps, called **normal forms**, each reducing redundancy and dependency by adhering to **certain rules**.

Key Normal Forms (1)

1. First Normal Form (1NF):

- Ensures that each column contains **atomic** (indivisible) values. No multi-valued attributes are allowed!

Example of a Table Not in 1NF

Student_ID	Name	Subjects
101	Alice	Math, Physics
102	Bob	Chemistry
103	Charlie	Math, CS

2. Second Normal Form (2NF):

- Must satisfy 1NF.
- Ensures that all non-key attributes are <u>fully</u> functionally dependent on the <u>entire</u> primary key (**No partial dependencies** exist)

Example of a Table That Violates 2NF

Student_ID	Course_ID	Student_Name	Course_Name	Instructor
101	CSE101	Alice	Databases	Dr. Smith
102	CSE101	Bob	Databases	Dr. Smith
103	CSE102	Charlie	Networks	Dr. Jones

Key Normal Forms (2)

3. Third Normal Form (3NF):

- Must satisfy 2NF.
- **No transitive dependencies** exist—non-key attributes <u>should not</u> depend on other non-key attributes.

Example of a Table That Violates 3NF

Student_ID	Name	Department_ID	Department_Name	HOD (Head of Department)
101	Alice	D01	Computer Science	Dr. Smith
102	Bob	D01	Computer Science	Dr. Smith
103	Charlie	D02	Mathematics	Dr. Johnson

Primary Key: `Student_ID`

4. Boyce-Codd Normal Form (BCNF):

- It is already in **Third Normal Form (3NF)**.
- For every functional dependency ($X \rightarrow Y$), X must be a **superkey** (i.e., X should uniquely determine all attributes of the table).

Functional Dependencies (FD)

- **FD** on a relation R states that "**if any two tuples in R agree (have same values) on the attributes $X_1, X_2, .., X_n$ then they also agree on the attributes $Y_1, Y_2, \ldots, Y_m$**

- The above FD is expressed as $X_1, X_2, .., X_n \longrightarrow Y_1, Y_2, \ldots, Y_m$

- Another representation is $X \rightarrow Y$

- A **FD implies a one-to-one (1:1) or many-to-one (M:1) relationship**, but **not** a one-to-many (1:M) relationship.

Keys of Relations

- Set of attributes are called key/candidate key (K) iff:
  - This set of attributes **functionally** determines **all other attributes.** No two tuples will agree on all attributes including K.
  - K must be minimal, meaning **no subset of K** can functionally determine all other attributes.

- *Set of attributes that* ***contain*** *key* K *is called* *superkey* for relation R*.* Superkey **satisfies** the first condition but not necessarily the second (minimality)

- K is a superkey but no subset of K is a superkey!

- Super keys help us find **candidate keys**, which in turn help define **primary keys**.

Inference Rules

- **Reflexivity**: if $B \subseteq A$, then $A \to B$ is trivial FD
- **Augmentation**: if $A \to B$, then $AC \to BC$
- **Transitivity**: if $A \to B$ and $B \to C$ then $A \to C$

Computing the Closure of Attributes

- **Closure algorithm:**
  - Start with: $Y^+ = Y$
  - Induction: Search for a **subset** of current Y^+ that **matches** the **LHS** of one of the FDs $(B_1, B_2, \ldots, B_m \rightarrow C)$ then add C to current Y^+
  - Repeat till nothing more can be added $\rightarrow Y^+$

Candidate key: Check all attributes that are **not part of the RHS** of our FDs and **minimal sets containing essential attributes** (Essential attributes are attributes that **must** be part of any candidate key). compute their closures, the attributes that determine <u>all attributes and minimal</u> are the Candidate keys.

Computing the Closure of Attributes – Example

Employee (Employee_ID, Name, Department_ID, Department_Name, Manager_ID, Manager_Name)

Functional Dependencies (FDs):

Employee_ID → Name, Department_ID
Department_ID → Department_Name, Manager_ID, Manager_Name
Manager_ID → Manager_Name

What is the Candidate Key?

$\{Employee\_ID\}^+$ = {Employee_ID, Name, Department_ID, Department_Name, Manager_ID, Manager_Name}

Boyce-Codd Normal Form - BCNF

- **BCNF** is a condition on a relation schema that eliminates potential **possible anomalies** by ensuring <u>all dependencies</u> are captured by **keys**.

- We say a relation *R* is in ***BCNF*** if whenever $X \rightarrow Y$ is a **nontrivial FD** that holds in *R*, then *X* is a **superkey**:
  - **Remember:** *nontrivial* means *Y* is not contained in *X*
  - **Remember**: a *superkey* is any superset of a key

Decomposition into BCNF

- **Input:** relation R_0 with FD's F_0
- ***Output:*** *decomposition of R_0 into set of relations R, all of which are in BCNF*
- $R = R_0$ & $F = F_0$
- Check if R is in BCNF; done.
- If there are BCNF violations, assume violation is $X \rightarrow Y$
- Compute closure of X: X^+
- Create two new relations: one with attributes X^+ and the other with the remaining attributes:
  - Choose $R_1 = X^+$
  - And $R_2 = R - (X^+ - X)$
  - Project FDs for each new relation
- Recursively decompose R_1 and R_2
- Return the union of all these decompositions

Decomposition: Is It All Good?

- A good decomposition should exhibits the following three properties:
  1. Elimination of anomalies
  2. Recoverability of information: recover original relation from the decomposed relations ← Lossless Join
  3. Preservation of Dependencies: if we joined the decomposed relations to reconstruct the original relation, do we satisfy the original FDs?
- ***BCNF*** decomposition gives us (1) and (2) but not always (3)!
- We will discuss later the ***third normal form (3NF)*** which will give us (2) and (3) but not necessarily (1)
- It is impossible to get all three at once!

3NF Let's Us Avoid Unenforceable FD

- 3rd Normal Form (3NF) modifies the BCNF condition so we do **not have** to decompose in this problem situation.
- An attribute is *prime* if it is a member of any key.
- $X \rightarrow A$ violates 3NF iff *X* is not a superkey, and also *A* is not prime.
- Relation R is in 3NF for nontrivial FD $X \rightarrow A$ iff: ***either*** X is a superkey ***or*** A's attributes are prime.

SQL Overview

SQL Statements

DML (Data Manipulation Language)	SELECT
	INSERT UPDATE DELETE
DDL (Data Definition Language)	CREATE ALTER DROP
DCL and Transaction Control	GRANT REVOKE COMMIT ROLLBACK

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
        <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Declaring Single-Attribute Keys

- Place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example:

```
CREATE TABLE Beers (
    name    CHAR(20) UNIQUE,
    manf    CHAR(20)
);
```

Example: Multi-attribute Key

- **The bar and beer together are the key for *Sells*:**

```
CREATE TABLE Sells (
        bar         CHAR(20),
        beer        VARCHAR(20),
        price       REAL,
        PRIMARY KEY (bar, beer)
);
```

PRIMARY KEY vs. UNIQUE

1. There can be **only one PRIMARY KEY** for a relation, but several UNIQUE attributes.
2. No attribute of a PRIMARY KEY can ever be **NULL** in any tuple. But attributes declared UNIQUE may have NULL's, and there may be several tuples with NULL.

Select-From-Where Statements

SELECT desired attributes

FROM one or more tables

WHERE condition about tuples of the tables

"*" In SELECT clauses

- When there is one relation in the FROM clause, * in the SELECT clause stands for "*all attributes of this relation*."
- Example: Using Beers(name, manf):

```
SELECT *
FROM Beers
WHERE manf = 'Anheuser-Busch';
```

Renaming Attributes

- If you want the result to have different attribute names, use "AS <new name>" to rename an attribute.

- Example: Using Beers(name, manf):

```
SELECT name AS beer, manf
FROM   Beers
WHERE  manf = 'Anheuser-Busch';
```

Expressions in SELECT Clauses

- Any expression that **makes sense** can appear as an element of a SELECT clause.
- Example: Using Sells(bar, beer, price):

```
SELECT bar, beer,
       price*114 AS priceInYen
FROM   Sells;
```

Complex Conditions in WHERE Clause

- Boolean operators AND, OR, NOT
- Comparisons =, <>, <, >, <=, >=
- And many other operators that produce boolean-valued results: BETWEEN, IN, LIKE, IS NULL, etc.

Patterns

- A condition can compare a string to a pattern by:

  <Attribute> LIKE <pattern> or

  <Attribute> NOT LIKE <pattern>

- *Pattern* is a quoted string with

  % = "any string."

  _ = "any single character."

Comparing NULL's to Values

- The logic of conditions in SQL is really 3- valued logic: TRUE, FALSE, UNKNOWN.
- Comparing any value (including NULL itself) with NULL yields UNKNOWN.
- A tuple is in a query answer iff the WHERE clause is TRUE (not FALSE or UNKNOWN).

Multi-table Queries

- Interesting queries often **combine data** from more than one table.
- We can address several tables in **one query** by listing them all in the FROM clause.
- **Distinguish attributes** of the <u>same name</u> in 2 tables by using “<table>.<attribute>”.

Example: Joining Two tables

- Using tables Emp(ename, dno) and Dept(dno, dname), find the department name of employee *Joe*.

```
SELECT dname
FROM   Emp, Dept
WHERE  ename = 'Joe' AND
        Emp.dno = Dept.dno;
```

Example: Self-Join

- From Beers(name, manf), find all pairs of beers by the same manufacturer.
  - Do not produce pairs like (Bud, Bud).
  - Produce pairs in alphabetic order, e.g. (Bud, Miller), not (Miller, Bud).

```
SELECT b1.name, b2.name
FROM   Beers b1, Beers b2
WHERE  b1.manf = b2.manf  AND
       b1.name < b2.name;
```

Sub-queries

- A parenthesized/nested SELECT-FROM-WHERE statement (*subquery*) can be used as a value in a number of places, including **FROM** and **WHERE** clauses.

- **Example:** in place of a table in the WHERE clause, we can use a subquery and then query its result.

The IN Operator

- <tuple> IN (<subquery>) is true if and only if the tuple is **a member** of the table produced by the subquery.
  - Opposite: <tuple> NOT IN (<subquery>).
- IN-expressions can appear in WHERE clauses.

The Exists Operator

- EXISTS(<subquery>) is true **if and only if** the subquery result is not empty.

- NOT EXISTS((<subquery>) is **true** if the subquery returns **no rows** (i.e., the result is empty).

Union, Intersection, and Difference

- Union, intersection, and difference of tables are expressed by the following forms, each involving subqueries:

  - (<subquery>) **UNION** (<subquery>)
  - (<subquery>) **INTERSECT** (<subquery>)
  - (<subquery>) **EXCEPT** (<subquery>)

Example: Find the names of all drinkers who either **like "Budweiser"** or have **bought "Budweiser"** at any bar.

DISTINCT

- From Sells(bar, beer, price), find all the **different** prices charged for beers:

```
SELECT DISTINCT  price
FROM    Sells;
```

- Notice that without DISTINCT, each price would be listed **as many times as** there were bar/beer pairs at that price.

Questions?

No class on Monday