Notes on CS-348:
Introduction to Database Management

Unversity of Waterloo

David Duan

Last Updated: May 26, 2021 (draft)

Contents

Chapter 1

Introduction

Section 1. Motivation

1.1. Consider part of a university organization that, among other data, keeps information about all instructors, students, departments, and course offerings. One way to keep the information on a computer is to store it in OS files. However, keeping organizational information in a file-processing system has a number of disadvantages:

- **Data redundancy and inconsistency:** As time passes, the same information may be duplicated in several files and the various copies of the same data may no longer agree.
- **Difficulty in accessing data:** The conventional file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner.
- **Data isolation:** Because data are scattered in various files which may be in different formats, writing new programs to retrieve appropriate data is difficult.
- **Integrity problems:** It is difficulty to enforce consistency constraints. The problem is compounded when constraints involve several data items from different files.
- **Atomicity problems:** Many operations must be *atomic* – it must happen in its entirety or not at all. It is difficulty to ensure atomicity in a conventional file-processing system.
- **Concurrent-access anomalies:** It is difficult to provide consistent query results when multiple clients are accessing and possibly modifying the data.
- **Security problems:** Not every user of the database system should be able to access all the data. Enforcing such security constraints is difficulty in a file-processing system.

In what follows, we shall see the concepts and algorithms that enable database systems to solve the problems with file-processing systems.

Section 2. Database Management Systems

2.1. Definition: A **database** is a large and persistent collection of data and metadata organized in a way that facilitates efficient retrieval and revision.

2.2. Definition: A **data model** is a collection conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

2.3. Note (Data Models): The data models can be classified into four different categories:

- **Relational Model.** Uses a collection of tables (known as **relations**) to represent both data and the relationships among those data.
  - Each table has multiple columns and each column has a unique name.
  - Each row of the table represents one piece of information.
- **Entity-Relationship Model.** Uses a collection of basic objects, called **entities**, and **relationships** among these objects. Widely used in database design.
- **Object-Based Data Model.** Extending the E-R model with encapsulation, methods, and object identity.
- **Semi-structured Data Model.** Permits the specification of data where individual data items of the same type may have different sets of attributes.
  - This is in contrast to the data models mentioned earlier, where every data item of a particular type must have the same set of attributes.
  - Examples include JSON and XML.

2.4. Note (Data Abstraction): Database developers hide the complexity from users through several levels of **data abstraction**, to simplify users' interactions with the system:

- **Physical level:** how the data are actually stored (on a physical level).
- **Logical level:** what data are stored and what relationship exists among those data.
- **View level:** provide different views for different users based on needs.

2.5. Definition: The collection of information stored in the database at a particular moment is called an **instance** of the database. The overall design of the database is called the database **schema**. The **physical schema** describes the database design at the physical level, while the **logical schema** describes the database design at a logical level. At the view level, a **subschema** describes a view of the database.

2.6. Definition: A **database management system** (DBMS) is a set of programs that implements a dta model to manage a database.

Section 3. Database Languages

3.1. Definition: A DBMS provides a **data-definition language** (DDL) to specify the database schema and a **data-manipulation language** (DML) to express database queries and updates.

3.2. Note: We specify the storage structure and access methods used by the DBMS by a set of statements in a special type of DDL, called a **data storage and definition** language. These statements define the implementation details of the database schemas, which are usually hidden from users.

3.3. Note: The data values stored in the database must satisfy certain **consistency constraints**. The DDL provides facilities to specify such constraints. The DBMS checks these constraints every time the database is updated.

- **Domain Constraints.** A domain of possible values must be associated with every attribute, e.g., integer types, character types, data-time types, etc.
- **Referential Integrity.** A value that appears in one relation for a given set of attributes also appear in a certain set of attributes in anther relation.
- **Authorization.** Give different users different access privilege, e.g., *read authorization*, *insert authorization*, *update authorization*, and *delete authorization*, etc.

3.4. Note: The processing of DDL statements generates some output, which is placed in the **data dictionary**, which contains **metadata**. The data dictionary is considered to be a special type of table that can be accessed and updated only by the DBMS itself (not a regular user). The DBMS consults the data dictionary before reading or modifying actual data.

3.5. Note: There are two types of DML:

- **Procedural DMLs** require a user to specify *what* data are needed and *how* to get them.
- **Declarative DMLs** (or **nonprocedural DMLs**) require a user to specify *what* data are needed *without* specifying how to get those data.

Declarative DMLs are usually easier as the user does not have to specify how to get the data. Instead, the DBMS will figure out an efficient means of accessing data.

3.6. Definition: A **query** is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a **query language**.

3.7. Definition: A **transaction** is a sequence of *indivisible* DML requests; applications access a database via transactions. The **ACID properties** of a transaction, which is short for *atomicity*, *consistency*, *isolation*, and *durability*, guarantees data integrity.

Chapter 2

The Relational Model

Section 1. Relational Databases

1.1. Motivation: Set-comprehension $\{(x_1, \ldots, x_n) : \text{condition}\}$ specifies the conditions that the variables $(x_1, \ldots, x_n)$ need to satisfy without specifying how to find them.

1.2. Example: Let $+(x, y, z)$ denote the relation $z = x + y$. We can express the $+$ table as

$$\left|\begin{array}{c|c|c} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 2 \\ \vdots & \vdots & \vdots \end{array}\right|$$

To find (x, y) such that $x + y = x - y$, note that $x - y = z \iff z + y = x$:

$$\{(x, y) \mid \exists z : +(x, y, z) \wedge +(z, y, x)\}.$$

To find the additive identity:

$$\{x : +(x, x, x)\} = \{0\}.$$

1.3. Note: In a **relational database**, all information is organized into a *finite* number of **relations** or **tables**. Features of relational databases include:

- simple and clean data model accommodating *data independence*;
- powerful and declarative *DML* based on well-formed formulas in first order predicate logic;
- *integrity constraints* via well-formed formulas.

1.4. Definition: There are three components to a relational database:

(1). **Universe**, a set of *values* $\mathbf{D}$ with equality $=$;

(2). **Relation**,

- Metadata: *predicate name* R (table name) and *arity* k (# of columns), written R/k;
- Extension: a set of k-tuples, $\mathbf{R} \subseteq \mathbf{D}^k$, i.e., an *instance* of the relation R.

(3). **Database**

- Signature (metadata): finite set $\rho = (R_1, \ldots, R_n)$ of predicate names;
- Instance (data, structure): an extension $\mathbf{R}_i$ for each relation R_i.

An **instance** of a relational db is $\mathbf{DB} = (\mathbf{D}, =, \mathbf{R}_1, \ldots, \mathbf{R}_n)$, where each $\mathbf{R}_i$ is an instance of R_i.

1.5. Example: The set of integers, together with addition and multiplication (represented by tables, see Example 1.2), can be viewed as a relational database, with signature $\rho := (\text{PLUS}/3, \text{TIMES}/3)$ and data $\mathbf{DB} := (\mathbb{Z}, =, \mathbf{PLUS}, \mathbf{TIMES})$.

1.6. Definition: A **valuation** is a function θ that maps variable names to values in the universe:

$$\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}.$$

To denote a modification to θ in which variable x is instead mapped to value v, one writes

$$\theta[x \mapsto v].$$

1.7. Definition: Given a signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, a set of variable names $\{x_1, x_2, \ldots\}$, and a set of constants $\{c_1, c_2, \ldots\}$, the **conditions** are formulas defined by the grammar

$$\varphi := R_i(x_{i,1}, \ldots, x_{i,k_i}) \mid x_i = x_j \mid x_i = c_j \mid \varphi_1 \wedge \varphi_2 \mid \exists x_i : \varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi_1.$$

1.8. Definition: The **free variables** of a formula φ, written $\mathrm{Fv}(\varphi)$, are defined as follows:

$$\begin{array}{ll}
\mathrm{Fv}\left(R\left(x_{i_1}, \ldots, x_{i_k}\right)\right) & = \{x_{i_1}, \ldots, x_{i_k}\}; \\
\mathrm{Fv}\left(x_i = x_j\right) & = \{x_i, x_j\}; \\
\mathrm{Fv}\left(x_i = c_j\right) & = \{x_i\}; \\
\mathrm{Fv}(\varphi \wedge \psi) & = \mathrm{Fv}(\varphi) \cup \mathrm{Fv}(\psi); \\
\mathrm{Fv}\left(\exists x_i : \varphi\right) & = \mathrm{Fv}(\varphi) - \{x_i\} \\
\mathrm{Fv}(\varphi \vee \psi) & = \mathrm{Fv}(\varphi) \cup \mathrm{Fv}(\psi); \\
\mathrm{Fv}(\neg\varphi) & = \mathrm{Fv}(\varphi).
\end{array}$$

A condition is called a **sentence** when it has no free variables.

1.9. Definition: The **truth** of a formula φ over a *signature* $\rho = (R_1/k_1, \ldots, R_n/k_n)$ is defined wrt a *database instance* $\mathbf{DB} = (\mathbf{D}, =, \mathbf{R}_1, \ldots, \mathbf{R}_n)$ and a *valuation* $\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}$ as follows:

- $\mathbf{DB}, \theta \models R_i\left(x_{i,1}, \ldots, x_{i,k_i}\right)$ if $\left(\theta\left(x_{i,1}\right), \ldots, \theta\left(x_{i,k_i}\right)\right) \in \mathbf{R}_i$
- $\mathbf{DB}, \theta \models x_i = x_j$ if $\theta\left(x_i\right) = \theta\left(x_j\right)$;
- $\mathbf{DB}, \theta \models x_i = c_j$ if $\theta\left(x_i\right) = c_j$;
- $\mathbf{DB}, \theta \models \varphi \wedge \psi$ if $\mathbf{DB}, \theta \models \varphi$ and $\mathbf{DB}, \theta \models \psi$;
- $\mathbf{DB}, \theta \models \exists x_i : \varphi$ if $\mathbf{DB}, \theta\left[x_i \mapsto v\right] \models \varphi$, for some $v \in \mathbf{D}$;
- $\mathbf{DB}, \theta \models \varphi \vee \psi$ if $\mathbf{DB}, \theta \models \varphi$ or $\mathbf{DB}, \theta \models \psi$;
- $\mathbf{DB}, \theta \models \neg\varphi$ if $\mathbf{DB}, \theta \not\models \varphi$.

1.10. Definition: A **query** in the relational calculus is a set comprehension of the form

$$\{(x_1, \ldots, x_k) \mid \varphi\},$$

where $\{x_1, \ldots, x_k\} = \mathrm{Fv}(\varphi)$ (are the free variables of φ). The **answers** to a query $\{(x_1, \ldots, x_k) \mid \varphi\}$ over a database instance $\mathbf{DB}$ is the relation that consists of the valuations applied to the tuple of variables that make the formula true wrt the database:

$$\{(\theta(x_1), \ldots, \theta(x_k)) \mid \mathbf{DB}, \theta \models \varphi\}.$$

Section 2. Integrity Constraints

2.1. Note: A database signature captures only the structure of relations; valid database instances must satisfy additional **integrity constraints** in the form of sentences over the given signature. Consider the set of integers with addition (as a table). Since the addition is commutative, we must have $\forall x, y, z : +(x, y, z) \implies +(y, x, z)$. It is also a total function, so $\forall x, y : \exists z : +(x, y, z)$. Common types of integrity constraints include:

- **Datatypes:** Values of a particular attribute belong to a prescribed data type.
- **Keys:** Values of attributes are unique among tuples in a relation.
- **Referential Integrity:** Values appearing in one relation must appear in another relation.
- **Disjointness:** Values cannot appear simultaneously in certain relations.
- **Coverage:** Values in a relation must appear in at least one of another set of relations.

2.2. Definition: Given a signature ρ, a table R occurring in ρ is a **view** when the relational database schema contains exactly one integrity constraint of the form

$$\forall x_1, \ldots, x_k : R(x_1, \ldots, x_k) \iff \varphi,$$

where $\{x_1, \ldots, x_k\} = \mathrm{Fv}(\varphi)$. This condition φ is called the **view definition** of R and R is said to **depend on** any table mentioned in φ. Note that no table occurring in a schema is allowed to depend on itself, either directly or indirectly (i.e., dependency graph is acyclic).

2.3. Definition: A **relational database schema** is a pair $\langle \rho, \Sigma \rangle$, where ρ is a signature and Σ is a finite set of integrity constraints that are sentences over ρ.

2.4. Definition: A **relational database** consists of a relational database schema $\langle \rho, \Sigma \rangle$ and an instance **DB** of its signature ρ.

2.5. Definition: A relational database is **consistent** iff for any integrity constraint $\varphi \in \Sigma$ and any valuation θ, we have $\mathbf{DB}, \theta \models \varphi$.

Section 3. Safety and Finiteness

3.1. Motivation: So far, we have seen that *databases* are relational structures, *queries* are set comprehensions with conditions as formulas in first order predicate logic, and *integrity constraints* are sentences in FOPL. Are there any remaining issues? Yes. Relational databases and relational calculus queries should have the following properties:

- The extension of any relation in a signature should be *finite*;
- Queries should be *safe*, i.e., their answers should be *finite* when database instances are finite.

3.2. Definition: A relational calculus query $\{(x_1, \ldots, x_k) \mid \varphi\}$ is **domain independent** when, for any pair of instances $\mathbf{DB}_1 = (\mathbf{D}_1, =\mathbf{R}_1, \ldots, \mathbf{R}_k)$ and $\mathbf{DB}_2 = (\mathbf{D}_2, =, \mathbf{R}_1, \ldots, \mathbf{R}_k)$ and any θ,

$$\mathbf{DB}_1, \theta \models \varphi \iff \mathbf{DB}_2, \theta \models \varphi.$$

3.3. Intuition: The following theorem says that *domain independence* and *finite database instances* together give safety (finiteness) of queries.

3.4. Theorem: *Let $(R_1, \ldots, R_k)$ be the signature of a relational database. Answers to domain independent queries contain only values that occur in the extension $\mathbf{R}_i$ of any relation R_i.*

Proof. Suppose the query $Q := \{\mathbf{x} : \varphi\}$ is domain-independent. If there exists $y \in Q \backslash (\mathbf{R}_1 \cup \cdots \cup \mathbf{R}_k)$, we can redefine the domain to be $\mathbf{D}' := \mathbf{D} \setminus \{y\}$ so that under this domain, $y \notin Q$. This contradicts the domain-independence of query Q as querying in $\mathbf{D}$ and $\mathbf{D}'$ yields different results. □

3.5. Theorem: *Satisfiability of relational calculus queries is undecidable.*

Proof. Recall the undecidable problem *post correspondence problem* (PCP): Given two lists $u_1, \ldots, u_n$ and $v_1, \ldots, v_n$ of two words over the same alphabet Σ with at least two symbols, find a sequence if indexes $i_1, \ldots, i_k$, $1 \leq i_j \leq n$, such that $u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$. Now do reduction from PCP.[1] □

3.6. Theorem: *Domain independence of relational calculus queries is undecidable.*

Proof. Consider queries of the form $Q := \{x : (x = x) \wedge \varphi\}$ where x does not appear in φ.

- If φ is unsatisfiable, then no valuation θ exists such that $\mathbf{DB}, \theta \models \varphi$ and thus $Q = \varnothing$ is domain independent.
- If φ is satisfiable, there exists a valuation φ such that the free variables in φ are assigned and make φ True. For this valuation θ, Q will contain all elements of domain $\mathbf{D}$ and therefore Q is domain-dependent.

Since proving domain-independence is equivalent to proving satisfiability, we conclude that this problem is undecidable. □

[1] P122 to P126, Chapter 6, of Foundations of Databases, Abiteboul et. al.

3.7. Definition: Given a database signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, a set of variable names $\{x_1, x_2, \ldots\}$, and a set of constants $\{c_1, c_2, \ldots\}$, **range-restricted conditions** are formulas defined by the grammar

$$\begin{aligned}
\varphi \;:=\; & R_i\left(x_{i,1}, \ldots, x_{i,k_i}\right) \\
\mid\; & \varphi_1 \wedge (x_i = x_j)\,, \text{ where } \{x_i, x_j\} \cap \mathrm{Fv}(\varphi_1) \neq \varnothing \text{ (Case 1)} \\
\mid\; & x_i = c_j \\
\mid\; & \varphi_1 \wedge \varphi_2 \\
\mid\; & \exists x_i \cdot \varphi_1 \\
\mid\; & \varphi_1 \vee \varphi_2, \text{ where } \mathrm{Fv}(\varphi_1) = \mathrm{Fv}(\varphi_2) \text{ (Case 2)} \\
\mid\; & \varphi_1 \wedge \neg\varphi_2, \text{ where } \mathrm{Fv}(\varphi_1) = \mathrm{Fv}(\varphi_2) \text{ (Case 3)}
\end{aligned}$$

A **range-restricted RC query** has the form $\{(x_1, \ldots, x_n) \mid \varphi\}$ where $\{x_1, \ldots, x_n\} = \mathrm{Fv}(\varphi)$ and that φ is a range-restricted condition. A query language for the relational model is **relationally complete** if the language is at least as expressive as the range-restricted RC.

3.8. Theorem: *Every range-restricted RC query is an RC query and is domain independent.*

Proof. Both claims follow by simple inductions on the form of a range restricted condition. □

3.9. Theorem: *Every domain independent RC query has an equivalent formulation as a range-restricted RC query.*

Proof. Let φ be a domain independent query. Restrict every variable in φ to the active domain, then express the active domain using a unary query over the database instance. □

3.10. Note (Complexity):

- Evaluation of every query terminates. Thus, relational calculus is not Turing complete.
- **Data complexity** in the size of the database, for a fixed query:
  - PTIME: solvable by a deterministic TM using a polynomial amount of time.
  - LOGSPACE: solvable by a deterministic TM using a log amount of (auxiliary) memory.
  - AC_0: constant time on polynomially many CPUs in parallel.
- **Combined complexity** in the size of the query and the database:
  - PSPACE: solvable by a deterministic TM using a polynomial amount of space.
  - can express NP-hard problems (e.g., SAT).

3.11. To summarize, the *query evaluation* problem, which finds all answers to a query in a finite database instance, is much easier than the *query satisfiability* problem, which determines whether there is a finite database instance for which the answer is non-empty.

3.12. Definition: A query $\{(x_1, \ldots, x_k) \mid \varphi\}$ **subsumes** a query $\{(x_1, \ldots, x_k) \mid \psi\}$ wrt a database schema Σ if

$$\{(\phi(x_1), \ldots, \phi(x_k)) \mid \mathbf{DB}, \theta \models \psi\} \subseteq \{(\phi(x_1), \ldots, \phi(x_k)) \mid \mathbf{DB}, \theta \models \varphi\}$$

for every database $\mathbf{DB}$ such that $\mathbf{DB} \models \Sigma$.

3.13. Remark: This is necessary for query simplification and is equivalent to proving

$$\left(\bigwedge_{\varphi_i \in \Sigma} \varphi_i \right) \implies (\forall x_1, \ldots, x_k : \psi \implies \varphi).$$

This is decidable for a fragments of relational calculus but undecidable in general.

3.14. Note: Since RC is not Turing-complete, there must be computable queries that cannot be written in RC. We here give some examples:

- Ordering, Arithmetic, String Operations
- Counting, Aggregation
- Reachability, Connectivity (Paths in Graph)

Chapter 3

Introduction to SQL

Section 1. Overview of SQL

1.1. The **Structured Query Language** (SQL) has several parts:

- **DDL**: for defining, modifying, and deleting relation schemas.
- **DML**: for querying, inserting, deleting, and modifying data in the database.
- **Integrity** (DDL): for specifying integrity constraints that data stored in the db must satisfy.
- **View Definition** (DDL): for defining views.
- **Transaction Control**: for specifying the beginning and end points of transactions.
- **Embedded SQL and Dynamic SQL**: for specifying how SQL statements can be embedded within general purpose programming languages.
- **Authorization** (DDL): for specifying access rights to relations and views.

Section 2. SQL Data Definition

2.1. (CREATE TABLE) The general form of the `create table` command is

```
create table r (
    A1 D1,
    A2 D2,
    ...,
    An Dn,
    <integrity constraint 1>,
    ...
    <integrity constraint k>);
```

where r is the name of the relation, each A_i is the name of an attribute/column in the schema of relation r, and D_i is the domain of attribute A_i; that is, D_i specifies the type of attribute A_i along with optional constraints that restrict the set of allowed values for A_i. SQL prevents any update to the database that violates these integrity constraints.

2.2. Example: We start with the following example, which defines three tables in the database:

```
create table AUTHOR (
aid integer not null,
name varchar(10) not null,
primary key (aid) )

create table PUBLICATION (
pubid integer not null,
title varchar(25) not null,
primary key (pubid) )

create table WROTE (
author integer not null,
publication integer not null,
primary key (author, publication),
foreign key (author) references AUTHOR,
foreign key (publication) references PUBLICATION )
```

In this code snippet, we see four column varieties of integrity constraints:

- data type constraints for each column/attribute;
- `not null` constraints;
- `primary key` constraints;
- `foreign key` constraints.

We will explain these first.

2.3. (Data Types) The basic **data types** of SQL:

- `integer`: 32-bit integer
- `smallint`: 16-bit integer
- `decimal(m,n)`: fixed decimal
- `float`: 32-bit IEEE float
- `char(n)`: character string of length n
- `varchar(n)`: variable length string of length at most n
- `date`: year/month/day
- `time`: hh:mm:ss.ss

Each type may include a special value called the **null** value, which indicates an absent value that may exist but be unknown or that may not exist at all. To prohibiting a column from having null values, add `not null` for that column in the `create` statement.

2.4. (Primary Key, Foreign Key)

- ```
  1  PRIMARY KEY (A1, ..., An)
  ```

  Attributes $A_1, \ldots, A_n$ form the primary key of the relation. These attributes are required to be *nonnull* and *unique*.

- ```
  1  FOREIGN KEY (Aj1, Aj2, .., Ajn) references s
  ```

  The values of attributes $A_{j_1}, A_{j_2}, \ldots, A_{j_n}$ for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.

2.5. (ALTER TABLE)

- To add an attribute A of type D to an existing relation r:

  ```
  1  ALTER TABLE r ADD A D;
  ```

- To drop an attribute A from a relation r:

  ```
  1  ALTER TABLE r DROP A;
  ```

2.6. (DROP TABLE, DELETE FROM)

- To remove relation r from an SQL database delete all relevant information:

  ```
  1  DROP TABLE r;
  ```

- To delete all data from r but retain the relation:

  ```
  1  DELETE FROM r;
  ```

Section 3. Conjunctive Queries

3.1. Motivation: The `SELECT` block allows formulation of conjunctive $\exists, \wedge$ queries of the form

$$\left\{ \texttt{<results>} \mid \exists\ \texttt{<unused>} : \left(\bigwedge \texttt{<tables>} \right) \wedge \texttt{<condition>} \right\}.$$

where `<results>` specifies values in the resulting tuples and `<unused>` are variables *not used* in `<results>`. The basic syntax is given as follows:

```
SELECT [DISTINCT] <results>
FROM              <tables>
WHERE             <condition>
```

The role of each clause is as follows:

- The `SELECT` clause is used to list the attributes desired in the result of a query.
- The `FROM` clause is a list of relations to be accessed in the evaluation of the query.
- The `WHERE` clause is a predicate involving attributes of the relation in the `FROM` clause.
- The `DISTINCT` keyword can be added after `SELECT` to force the elimination of duplicates.

3.2. Intuition: The easiest way to understand this operation is to consider the clauses in operational order: first `FROM`, then `WHERE`, finally `SELECT`:

(1). Generate a Cartesian product of the relations listed in the `FROM` clause.

(2). Apply the predicates specified in the `WHERE` clause on the result of Step 1.

(3). For each tuple in the result of Step 2, output the attributes (or results of expressions) specified in the `SELECT` clause.

3.3. Example: Retrieve the names of all instructors, along with their department names and department building name, whose salary is greater than 70k.

```
SELECT name, instructor.dept_name, building
FROM instructor, department
WHERE instructor.dept_name = department.dept_name AND salary > 70000
```

A few remarks:

- Since `dept_name` appears in two tables, we use `instructor.dept_name` to make clear which attribute we are referring. In general, the syntax is `<table>.<attribute>`.
- In contrast, `name`, `building`, and `salary` appear in only one table, so there's no need to prepend the relation name.
- SQL allows the use of logical connectives `AND`, `OR`, and `NOT` as well as the comparison operators `<`, `<=`, `>`, `>=`, `=`, `<>` in the `WHERE` clause. More on this later.

3.4. (AS)

```
old_name AS new_name
```

The `AS` clause can appear in both `SELECT` and `FROM` clauses, which allows us to rename the attributes of a result relation and to rename relations. In the following two examples, we demonstrate two use cases of `AS`:

- Replace a long relation name with a shortened version for convenience.
- Compare tuples in the same relation.

3.5. Example: For all instructors in the university who have taught some course, find their names and the course ID of all courses they taught.

```
SELECT T.name, S.course_id
FROM instructor AS T, teaches AS S
WHERE T.ID = S.ID
```

3.6. Example: For the names of all instructors whose salary is greater than at least one instructor in the Biology department.

```
SELECT DISTINCT T.name
FROM instructor AS T, instructor AS S
WHERE T.salary > S.salary AND S.dept_name = 'Biology';
```

3.7. Remark: In the above query, T and S can be thought of as copies of the relation `instructor`, but more precisely, they are declared as *aliases* for the relation. An identifier, such as T and S, that is used to rename a relation is referred to as a **correlation name** in the SQL standard, but is also commonly referred to as a *table alias*, *correlation variable*, or *tuple variable*.

3.8. Note (String Operations):

- SQL specifies strings by enclosing them in single quotes. A single quote character that is part of a string can be specified by using two single quote characters.
- Common operations: length, upper/lower case conversion, concatenation, trimming, etc.
- Pattern matching via `LIKE`: `%` (percent) matches any substring; _ (underscore) matches any character; \ (black slash) to escape special characters.

3.9. (ORDER BY)

```
ORDER BY <attribute> [DESC/ASC]
```

The **order by** clause causes the tuples in the result of a query to appear in sorted order. Note this clause has to appear after `FROM` and `WHERE`.

3.10. (FROM)

```
FROM R1[[AS] n1], ..., Rk[[AS] nk]
```

- R_i are relation names and n_i are distinct identifiers.
- The clause represents a conjunction $R_1 \wedge \cdots \wedge R_k$.
- The clause can appear only as a part of the `SELECT` block (i.e., cannot appear alone).

3.11. (SELECT)

```
SELECT DISTINCT e1 [[AS] n1], ..., ek [[AS] nk]
```

Operates as follows:

(1). Eliminate superfluous attributes and remaining duplicates from answers.

(2). Evaluate expressions e_i (here, built-in functions can be applied to values of attributes).

(3). Give names n_i to expression values in the answer.

Use `SELECT *` if you want to retrieve all attributes from the results.

3.12. (WHERE)

```
WHERE <condition>
```

Additional conditions on tuples that qualify for the answer. We now look at the possible predicates:

- `BETWEEN`: to simplify the condition `v1 <= x AND x >= v2`.
- **Row constructor**: `(a1, a2) <= (b1, b2)` is equivalent to `a1 <= b1 AND a2 <= b2`.

Section 4. Set Operations

4.1. Motivation: So far, we have seen how the `SELECT` statement allows us to express $\exists$ and $\wedge$. In this section, we will express $\vee$ and $\neg$ with set operations and rewrite $\forall$ using negation and $\exists$.

4.2. Note: SQL provides `UNION`, `EXCEPT`, and `INTERSECT` for **set operations** $Q_1 \cup Q_2$, $Q_1 \setminus Q_2$, and $Q_1 \cap Q_2$. Note that operands in a set operation must have **union-compatible** signatures.

4.3. Example: These set operations automatically eliminates duplicates in the inputs before performing set difference. To retain duplicates, we must add `ALL` after each operation, e.g.,

```
(SELECT course_id FROM section WHERE year=2017 AND semester='Fall')   -- Q1
UNION ALL
(SELECT course_id FROM section WHERE year=2018 AND semester='Spring') -- Q2
```

4.4. To use a set operation inside a `SELECT` block, we can use named queries and inline queries.

- Recall that queries denote relations. SQL provides a naming mechanism to assign names to (results) of queries, which can be used later in place of (base) relations.

```
WITH T1 [<opt-schema-1>] AS ( <query-1-goes-here> ),
     ...
     Tn [<opt-schema-n>] AS ( <query-n-goes-here> )
<query-that-uses-T1-to-Tn-as-table-names>
```

- If we only use the query result once, then we can use **inline queries** in the `FROM` clause:

```
FROM ..., ( <query-here> ) AS <id>, ...
```

  This `<id>` stands for the result of `<query-here>` and is mandatory.

4.5. Example: List all publication titles for books or journals.

```
WITH bookorjournal (pubid) AS
    ((SELECT DISTINCT pubid FROM book) UNION (SELECT DISTINCT pubid FROM journal))
SELECT DISTINCT title
FROM publication, bookorjournal
WHERE publication.pubid = bookorjournal.pubid;
```

```
SELECT DISTINCT title
FROM publication,
    ((SELECT DISTINCT pubid FROM journal)
     UNION
     (SELECT DISTINCT public FROM book)) as journalorbook
WHERE publication.public = journalorbook.pubid;
```

Section 5. Nested Queries

5.1. SQL allows conditions in a `WHERE` clause to be expressed with subqueries, which simplifies writing queries with negations and can make code more readable. However, this leads to more complicated semantics (particularly when duplicates are involved) and is very error-prone.

5.2. Note:

- Presence/absence of a *single value* in a subquery:

```
<attr> IN ( <query> ); <attr> NOT IN ( <query> );
```

- Relation of a value to some/all values in a subquery:

```
<attr> op SOME ( <query> ); <attr> op ALL ( <query> );
```

- Emptiness/non-emptiness of a subquery:

```
EXISTS ( <query> ); NOT EXISTS ( <query> );
```

In the first two cases, `<query>` must be unary.

5.3. Example: Get titles of all articles.

```
SELECT DISTINCT title
FROM publication
WHERE pubid IN ( SELECT pubid FROM article )
```

5.4. Example: Find the longest book.

```
SELECT DISTINCT title
FROM books
WHERE endpage-startpage >= all ( SELECT endpage-startpage FROM books )
```

Note how we used the binary operator $-$ in the `WHERE` clause and the subquery.

5.5. Example: Find all students who have taken all courses offered by the math department.

```
SELECT DISTINCT S.ID, S.name
FROM student as S  -- renaming the relation
WHERE NOT EXISTS
    ((SELECT course_id FROM course WHERE dept_name = 'MATH')
     EXCEPT
     (SELECT T.course_id FROM takes as T WHERE S.ID = T.ID))
```

5.6. Remark: Nesting in the `WHERE` clause is mere syntactic sugar:

```
SELECT r.b
FROM r
WHERE r.a in ( SELECT b FROM s )
```

```
SELECT r.b
FROM r, (SELECT DISTINCT b FROM s) as s
WHERE r.a = s.b
```

5.7. So far, subqueries have been independent from the main query. SQL also allows *parametric* subqueries, where the `<query>` mentions attributes in the main query. The *truth* of a predicate defined by a subquery is determined for each substitution (tuple) in the main query: instantiate all the parameters used, then check for truth value as before.

5.8. Example: Publications of at least two authors.

```
SELECT *
FROM wrote AS r
WHERE EXISTS (
    SELECT *
    FROM wrote AS s
    WHERE r.publication = s.publication AND r.author <> s.author
);
```

Equivalently, we can write

```
SELECT *
FROM wrote AS r
WHERE publication IN (
    SELECT publication
    FROM wrote AS r
    WHERE r.author <> s.author
);
```

Expressing negation is also easy; just add `NOT` before `EXISTS` and `IN`, respectively.

5.9. Since `WHERE` subqueries are also queries, one can nest repeatedly to form very complex search conditions. Every nested subquery can use attributes from the enclosing queries as parameters. However, attributes present in the subqueries only cannot be used to construct the results.

5.10. To summarize, `WHERE` subqueries enable easy formulation of queries of the form

$$\text{All } x \text{ in } R \text{ such that (a part of) } x \text{ does not appear in } S.$$

Note that subqueries only stand for `WHERE` conditions and cannot be used to produce results. You can use input parameters but these must be bounded in the main query.

Section 6. Aggregation

6.1. Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five standard built-in aggregate functions: `AVG`, `MIN`, `MAX`, `SUM`, `COUNT`. The input to `SUM` and `AVG` must be a collection of numbers, while the other operators can operate on collections of non-numeric data types, such as strings, as well.

6.2. Example: Find the average salary of instructors in the math department.

```
SELECT AVG(salary) AS avg_salary
FROM instructor
WHERE dept_name = 'math';
```

6.3. (GROUP BY) Suppose we would like to apply the aggregate function not only to a single set of tuples, but also to a group of sets of tuples. We can specify this in SQL using the `GROUP BY` clause. The attribute(s) given in the `GROUP BY` clause are used to form groups. Tuples with the same value on all attributes in the `GROUP BY` clause are placed in one group.

```
SELECT x1, ..., xk, agg1 [[AS] n1], ..., aggj [[AS] nj]
<FROM-WHERE>
[GROUP BY x1, ..., xk]
```

Note that all attributes in the `SELECT` clause that are not in the scope of an aggregate function must appear in a `GROUP BY` clause. These attributes are used for partitioning.

6.4. Example: Find the average salary of instructors in each department.

```
SELECT dept_name, AVG(salary) AS avg_salary
FROM instructor
GROUP BY dept_name;
```

6.5. Example: For each publication, count the number of authors.

```
SELECT publication, COUNT(author) FROM wrote GROUP BY publication
```

6.6. More explicitly, the aggregate functions operate as follows:

(1). Partition the result of `<FROM-WHERE>` into groups with equal values of `GROUP BY` attributes.

(2). On each of these partitions, apply the aggregation functions.

(3). For each group, add a tuple with the grouping attribute values and the results of the aggregate functions to the result.

It is always a good idea to name the results of the aggregate functions in the `SELECT` clause.

6.7. (HAVING) The `WHERE` clause cannot impose conditions on values of aggregates as `WHERE` conditions are applied *before* `GROUP BY`. SQL introduces a `HAVING` clause for this purpose, which is like `WHERE`, but for aggregate values. The aggregate functions used in the `HAVING` clause may be different from those in the `SELECT` clause, but the grouping is common. As with the case for `SELECT`, any attribute that is present in the `HAVING` clause without being aggregated must appear in the `GROUP BY` clause.

6.8. Example: List all publications with more than one author.

```
1  SELECT publication, COUNT(author) as acnt
2  FROM wrote
3  GROUP BY publication
4  HAVING COUNT(author) > 1
```

6.9. Note: The meaning of a query containing aggregation, `GROUP BY`, or `HAVING` clauses is defined by the following sequence of operations:

(1). Evaluate `FROM` first to get a relation.

(2). If `WHERE` is present, apply the conditions on the relation from `FROM`.

(3). Tuples satisfying the `WHERE` predicate(s) are then placed into groups by the `GROUP BY` clause if it is present. Otherwise, the entire set of tuples satisfy the `WHERE` predicate(s) is treated as being in one group.

(4). The `HAVING` clause, if present, is applied to each group; the groups that do not satisfy the `HAVING` clause predicate(s) are removed.

(5). The `SELECT` clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

Section 7. Transactions and Database Update

7.1. There are three kinds of table updates:

- `INSERT`, for inserting a single constant tuple or each tuple in the result of a query;
- `DELETE`, for removing all tuples satisfying a condition;
- `UPDATE`, for updating *in-place* all tuples satisfying a condition.

7.2. (INSERT) Add a tuple $(c_1, \ldots, c_k)$ to table T (c_i matches the type for attribute A_i):

```
INSERT INTO T[(A1, ..., Ak)] VALUES (c1, ..., ck)
```

To insert multiple tuples computed by query Q to table T:

```
INSERT INTO T (Q)
```

7.3. Example: A simple insert:

```
INSERT INTO author (aid, name) VALUES (4, 'yyk')
```

Add Tim as an author, with a new unique identification:

```
INSERT INTO author
    (SELECT max(aid) + 1, 'TIM'   -- subquery returns max(aid)+1 and 'TIM'
     FROM author)
```

7.4. (DELETE) Deletion all tuples that match `<condition>`.

```
DELETE FROM T WHERE <condition>
```

7.5. Example: Delete all authors who have not written anything.

```
DELETE FROM author
WHERE NOT EXISTS
    (SELECT * FROM wrote WHERE author = aid)
```

7.6. (UPDATE) Search in table T for all tuples that match `<condition>`, then update their values specified by `<assignments>`.

```
UPDATE T
SET <assignments>
WHERE <condition>
```

7.7. Example: Update anyone named Sue to be named Susan instead.

```
1 UPDATE author
2 SET name = 'Susan'
3 WHERE aid in (SELECT aid FROM author WHERE name = 'Sue')
```

7.8. A **transaction** consists of a sequence of query and/or update statements. The SQL standard specifies that a transaction begins implicitly when an SQL statement is executed. To signal an end of the transaction,

- `COMMIT` commits the current transaction, i.e., it makes the updates performed by the transaction become permanent in the database. After the transaction is committed, a new transaction is automatically started.
- `ROLLBACK` discards changes, i.e., it undoes all the updates performed by the SQL statements in the transaction. Thus, the database state is restored to what it was before the first statement of the transaction was executed.

Once a transaction has executed `COMMIT`, its effects can no longer be undone by `ROLLBACK`. The database system guarantees that in the event of some failure, such as a error in one of the SQL statements, a power outage, or a system crash, a transaction's effects will be rolled by if it has not yet executed `COMMIT`.

By either committing the actions of a transaction after all its steps are completed, or rolling back all its actions in case the transaction could not complete all its actions successfully, the database provides an abstraction of a transaction as being atomic, that is, indivisible. Either all the effects of the transaction are reflected in the database or none are (after rollback).

Chapter 4

More on SQL

Section 1. General Integrity Constraints

1.1. (CHECK) When applied to a relation declaration, the clause `CHECK(<condition>)` specifies a predicate that must be satisfied by every tuple in the relation. A common use of this is to ensure that attribute values satisfy specified conditions. It may appear on its own, or as part of the declaration of an attribute.

1.2. Example: Put `CHECK` on its own vs as part of the declaration of an attribute:

```
CREATE TABLE emp (
    ssn INTEGER NOT NULL,
    name CHAR(20),
    salary DEC(8, 2),
    PRIMARY KEY (ssn),
    CHECK (salary > 0) )
```

```
CREATE TABLE emp (
    ssn INTEGER NOT NULL,
    name CHAR(20),
    salary DEC(8, 2) CHECK (salary > 0),
    PRIMARY KEY (ssn)
)
```

1.3. (Assertion) An **assertion** is a predicate expressing a condition that we wish the database always to satisfy. When an assertion is created, the system tests for its validity. If valid, any future modification to the database is allowed only if it does not cause the assertion to be violated. If any condition fails during a transaction, a rollback will be triggered.

```
CREATE ASSERTION <assertion-name>
CHECK (<condition>)
```

1.4. Example: No pair of publications have the same `pubid`.

$$\forall p, t_1, t_2 : (\texttt{pub\_id}(p, t_1) \land \texttt{pub\_id}(p, t_2) \implies t_1 = t_2).$$

```
CREATE ASSERTION unique-pubid
CHECK (
    NOT EXISTS (
        SELECT * FROM publication AS p1, publication AS p2
        WHERE p1.pubid = p2.pubid AND p1.title != p2.title))
```

1.5. Example: Every `author` value in a `wrote` tuple occurs as an `aid` value of some tuple in the `author` table.

```
CREATE ASSERTION author-foreign-key
CHECK (
    NOT EXISTS (
        SELECT * FROM wrote AS w
        WHERE NOT EXISTS (
            SELECT * FROM author AS a WHERE a.aid = w.author)))
```

Section 2. Views

2.1. (Views) SQL allows a *virtual relation* to be defined by a query which conceptually contains the result of the query. This virtual relation, called a **view**, is not precomputed and stored but instead is computed by executing the query whenever the virtual relation is used. The following statement creates a view named v that stores the results from `<query>`:

```
1  CREATE VIEW v AS <query>;
```

View names may appear in a query any place where a relation name may appear. In particular, one view may be used in the expression defining another view.

2.2. (Materialized Views) Certain DBMS allow view relations to be stored, but ensure that if the actual relations used in the view definition change, the view is kept up-to-date. Such views are called **materialized views.** The process of keeping the materialized view up-to-date is called **materialized view maintenance**, which can be done eagerly, lazily, or periodically, depending on the DBMS design.

2.3. (Updatable) Modifications are generally not permitted on view relations, except in limited cases. In general, an SQL view is said to be **updatable** if the following conditions are all satisfied by the query defining the view:

- The `FROM` clause has only one database relation.
- The `SELECT` clause contains only attribute names of the relation and does not have any expressions, aggregates, or `DISTINCT` specification.
- Any attribute not listed in the `SELECT` clause can be set to null, i.e., it does not have a `NOT NULL` constraint and is not part of a primary key.
- The query does not have a `GROUP BY` or `HAVING` clause.
- In addition, if a `CHECK` clause is present, then the update must satisfy the condition.

Section 3. On Multiset Semantics

3.1. SQL has a more general **multiset** or bag semantics that allows duplicates, in contrast to the simpler set semantics of RC. More explicitly, each $R/k \in \rho$ (table R with arity k in the schema ρ) has an extra (hidden) column that represents a *count* of the number of occurrences of a tuple in any extension of R. No direct access to the repetition counts is possible, but you can infer this by inspecting the relation itself.

3.2. A multiset semantics is assumed for existential quantification in the RC query. An additional kind of condition "$\{\cdot\}$" removes duplicates. Let us update the definitions from Chapter 2.

3.3. Definition: Given a database signature $\rho = (R_1/k_1, \ldots, R_n/k_n)$, a set of variable names $\{x_1, x_2, \ldots\}$, and a set of constants $\{c_1, c_2, \ldots\}$, **range-restricted conditions** are formulas defined by the grammar

$$\begin{array}{rcl}
\varphi & := & R_i\left(x_{i,1}, \ldots, x_{i,k_i}\right) \\
& \mid & \varphi_1 \wedge \left(x_i = x_j\right), \text{ where } \{x_i, x_j\} \cap \mathrm{Fv}(\varphi_1) \neq \varnothing \text{ (Case 1)} \\
& \mid & x_i = c_j \\
& \mid & \varphi_1 \wedge \varphi_2 \\
& \mid & \exists x_i \cdot \varphi_1 \\
& \mid & \varphi_1 \vee \varphi_2, \text{ where } \mathrm{Fv}(\varphi_1) = \mathrm{Fv}(\varphi_2) \text{ (Case 2)} \\
& \mid & \varphi_1 \wedge \neg\varphi_2, \text{ where } \mathrm{Fv}(\varphi_1) = \mathrm{Fv}(\varphi_2) \text{ (Case 3)} \\
& \mid & \{\varphi\} \text{ (the } \textbf{duplicate elimination operator}\text{)}
\end{array}$$

A **range-restricted RC query** has the form $\{(x_1, \ldots, x_n) \mid \varphi\}$ where $\{x_1, \ldots, x_n\} = \mathrm{Fv}(\varphi)$ and that φ is a range-restricted condition.

3.4. Definition: A formula φ over a signature $\rho = (R_1/k_1, \ldots R_n/k_n)$ is **true** m **times** wrt database instance $\mathbf{DB} = (\mathbf{D}, =, \mathbf{R}_1, \ldots, \mathbf{R}_n)$ and valuation $\theta : \{x_1, x_2, \ldots\} \to \mathbf{D}$, written $\mathbf{DB}, \theta, m \models \varphi$, as given by the following:

$$\begin{array}{ll}
\mathbf{DB}, \theta, 0 \models R_i\left(x_{i,1}, \ldots, x_{i,k_i}\right) & \text{if } \left(\theta\left(x_{i,1}\right), \ldots, \theta\left(x_{i,k_i}\right), m\right) \notin \mathbf{R}_i, \text{ for any } m > 0 \\
\mathbf{DB}, \theta, m \models R_i\left(x_{i,1}, \ldots, x_{i,k_i}\right) & \text{if } \left(\theta\left(x_{i,1}\right), \ldots, \theta\left(x_{i,k_i}\right), m\right) \in \mathbf{R}_I \\
\mathbf{DB}, \theta, m \models \varphi \wedge \left(x_i = x_j\right) & \text{if } \mathbf{DB}, \theta, m \models \varphi \wedge \theta\left(x_i\right) = \theta\left(x_j\right) \\
\mathbf{DB}, \theta, 1 \models \left(x_i \approx c_j\right) & \text{if } \theta\left(x_i\right) = c_j \\
\mathbf{DB}, \theta, m_1 \cdot m_2 \models \varphi \wedge \psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \wedge \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, \sum_{v \in D} m_v \models \exists x \cdot \varphi & \text{if } \mathbf{DB}, \theta[x := v], m_v \models \varphi \\
\mathbf{DB}, \theta, m_1 + m_2 \models \varphi \vee \psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \wedge \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, \max\left(0, m_1 - m_2\right) \models \varphi \wedge \neg\psi & \text{if } \mathbf{DB}, \theta, m_1 \models \varphi \wedge \mathbf{DB}, \theta, m_2 \models \psi \\
\mathbf{DB}, \theta, 1 \models \{\varphi\} & \text{if } \mathbf{DB}, \theta, m \models \varphi
\end{array}$$

The **answers** to a query $\{(x_1, \ldots, x_n) \mid \varphi\}$ over $\mathbf{DB}$ is given as follows, when $\{x_1, \ldots, x_n\} = \mathrm{Fv}(\varphi)$:

$$\left\{\left(\theta\left(x_1\right), \ldots, \theta\left(x_n\right), m\right) \mid m > 0 \wedge \mathbf{DB}, \theta, m \models \varphi\right\}$$

Section 4. Null Values

4.1. The keyword **null** indicates an absence of value.

- Arithmetic: The result of an arithmetic expression evaluates to null if any input is null.
- Comparison: SQL introduces `UNKNOWN` to represent any comparison involving a null value.
  - In particular, `NULL = NULL` evaluates to `UNKNOWN`.
- Logic operations. Let U denote `UNKNOWN`.
  - AND: $T \wedge U = U$; $F \wedge U = F$; $U \wedge U = U$.
  - OR: $T \vee U = T$; $F \vee U = F$; $U \vee U = U$.
  - NOT: $\neg U = U$.
- Predicates: `IS NULL`, `IS NOT NULL`, `IS UNKNOWN`.

4.2. Remark: When a query uses the select distinct clause, duplicate tuples must be eliminated. For this purpose, when comparing values of corresponding attributes from two tuples, the values are treated as identical if either both are non-null and equal in value, or both are null. The approach of treating tuples as identical if they have the same values for all attributes, even if some of the values are null, is also used for the set operations union, intersection, and except.

4.3. Example: List all author ids and names for which their homepage is unknown.

```
1  SELECT aid, name
2  FROM author
3  WHERE uri IS NULL;
```

Section 5. Ordering and Limits

5.1. No particular ordering on the rows of a table can be assumed when queries are written. No particular ordering of rows of an intermediate result in the query can be assumed either. However, it is possible to order the final result of a query with an `ORDER BY` clause at the end of the query.

```
ORDER BY e1 [Dir1], ..., ek [Dirk]
```

`Dir` is either `ASC` (default) or `DESC`.

5.2. Example: List all authors in the database in ascending order of their name.

```
SELECT DISTINCT *
FROM author
ORDER BY name;
```

5.3. The number of results of a query can be **limited** by appending a `LIMIT` clause to a query.

```
<query> LIMIT e1 [ OFFSET e2 ];
```

Here, e_i's are numeric expressions. Note this semantics is *non-deterministic*:

- As many of the first e_1 answers to a query that exist for *any total order* to which the results of `<query>` can be extended, if there is no `OFFSET` given.
- As many of the next e_1 answers to a query that exist following the first e_2 answers for *any total order* to which the results of `<query>` can be extended, if `OFFSET` is given.

Semantics becomes deterministic only when `<query>` has an `ORDER BY` clause inducing a total order.

5.4. Example: List at most the first two entries of a sorted list of authors by name.

```
SELECT DISTINCT *
FROM author
ORDER BY name
LIMIT 2
```

List at most the next two entries following the second entry of a sorted list of authors by name.

```
SELECT DISTINCT *
FROM author
ORDER BY name
LIMIT 2 OFFSET 3
```

Section 6. Triggers

6.1. A **trigger** is a statement that the system executes automatically as a side effect of a modification to the database. To define a tigger, we must:

- specify when a trigger to be executed, which has two parts:
  - an *event* that causes the trigger to be checked, and
  - a *condition* that must be satisfied for trigger execution to proceed;
- specify the **actions** when the trigger executes.

```
CREATE TRIGGER <trigger-name>
AFTER <event> ON <table-or-view>
    [ REFERENCING OLD AS <corelation-old> ]
    [ REFERENCING NEW AS <corelation-new> ]
    FOR EACH ROW
    [ WHEN <condition> ]
    BEGIN ATOMIC
        <DML-action-1>;
        ...
        <DML-action-n>;
END
```

Here, `<event>` can be `INSERT`, `DELETE`, or `UPDATE ON <attribute>`.

6.2. Example: To main the number of credits earned by each student:

```
CREATE TRIGGER credits_earned
AFTER UPDATE OF takes ON grade
    REFERENCING NEW ROW AS nrow
    REFERENCING OLD ROW AS orow
    FOR EACH ROW
    WHEN nrow.grade <> 'F' AND nrow.grade IS NOT NULL
        AND (orow.grade = 'F' OR orow.grade IS NULL)
    BEGIN ATOMIC
        UPDATE student
        SET tot_cred = tot_cred +
            (SELECT credits FROM course
             WHERE course.course_id = nrow.course_id)
        WHERE student.id = nrow.id;
    END;
```

6.3. Triggers in SQL implements **event/condition/action** (ECA) rules and make the SQL DML Turing-complete. Note that the timing of integrity constraint checking must be carefully managed. Deferring all to `COMMIT` time always works but may impact performance.

6.4. Special syntax exists for common rules related to foreign key constraints:

```
FOREIGN KEY ( <from-attribute-list> )
REFERENCES <table> [ ( <to-attribute-list> ) ]
    [ ON DELETE <action> ]
    [ ON UPDATE <action> ]
```

where `<action>` can be:

- `RESTRICT`: produce an error;
- `CASCADE`: propagate the delete;
- `SET NULL`: set to "unknown".

6.5. Example:

```
CREATE TABLE wrote (
    author INTEGER NOT NULL,
    publication INTEGER NOT NULL,
    PRIMARY KEY (author, publication),
    FOREIGN KEY (author) REFERENCES author
        ON DELETE CASCADE,
    FOREIGN KEY (publication) REFERENCES publication )
```

Section 7. Authorization

7.1. The SQL DML includes a **data control language** (DCL) to manage access rights to database objects by users and groups. Each user may be authorized of all, none, or a combination of the following **privileges** on a specified parts of a database, such as a relation or a view: *read*, *insert*, *update*, and *delete*.

When a user submits a query or an update, the SQL implementation first checks if the query or update is authorized, based on the authorizations that the user has been granted. If the query or update is not authorized, it is rejected. In addition to authorizations on data, users may also be granted authorizations on the database schema, allowing them, for example, to create, modify, or drop relations.

The ultimate form of authority is that given to the database administrator. The database administrator may authorize new users, restructure the database, and so on. This form of authorization is analogous to that of a superuser, administrator, or operator for an operating system.
Here's the simplified syntax:

```
GRANT  <role> TO <user>
GRANT  <what> ON <object> TO <user-or-role>
REVOKE <role> FROM <user>
REVOKE <what> ON <object> FROM <user-or-role>
```

where `<what> ON <object>` can be

- For databases: `CONNECT`.
- For tables or views: `ALTER`, `REFERENCES`, `SELECT`, `INSERT`, `DELETE`, or `UPDATE`.

To create new roles, use

```
CREATE ROLE <role>.
```

The role `DBADM` is a superuser.

7.2. Example: Create the `pat` role and grant ability to access the `payroll` database to `pat`.

```
CREATE ROLE pat;
GRANT CONNECT ON payroll TO pat;
```

Grant ability to query table `employee` to `pat`:

```
GRANT SELECT ON employee TO pat;
```

Add yyk to the payroll project team:

```
GRANT pat TO yyk;
```

Chapter 5

The Entity-Relationship Data Model

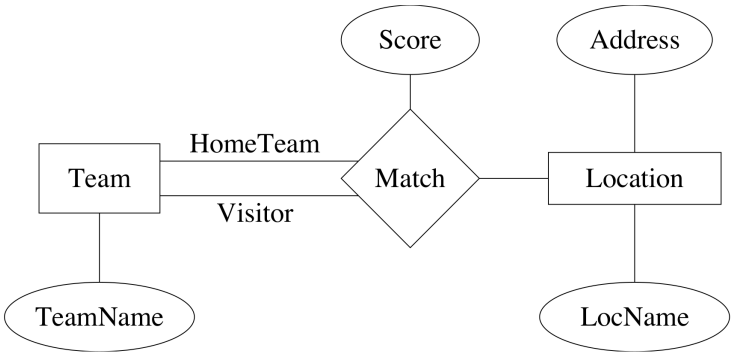
Section 1. The Entity-Relationship Model

1.1. A **conceptual data model** serves as a first step in formally capturing the metadata form information systems. Informal requirements for the underlying information are mapped to such a model. In designing a database schema, we must ensure that we avoid two major pitfalls: *redundancy* and *incompleteness*.

1.2. The **entity-relationship** (E-R) data model was developed to facilitate database design by allowing specification of an enterprise schema that represents the overall logical structure of a database. The E-R model employs three basic concepts: **entities**, **relationships**, and **attributes**.

- Entity:
  - An **entity** is a *distinguishable* thing.
  - An **entity set** is a set of entities of the same variety.
  - An entity set is represented in an E-R diagram by a *rectangle*.
- Attribute:
  - An **attribute** describes properties of entities.
  - The **domain** of an attribute is the set of permitted values for this attribute.
  - An attribute set is represented in an E-R diagram by a *ellipse*.
- Relationship:
  - A **relationship** is an association among several entities.
  - A **relationship set** is a set of relationships of the same type.
  - An relationship set is represented in an E-R diagram by a *rhombuses*.
- Role:
  - A **role** is the function of an entity set in a relationship set.
  - A **role name** is an explicit indication of a role.
  - Role labels are needed when an entity set has multiple functions in a relationship set.

1.3. Example:



Section 2. Integrity Constraints

2.1. There are four varieties of integrity constraints in an ER model that are commonly expressed with graphical annotations:

(1). primary keys;

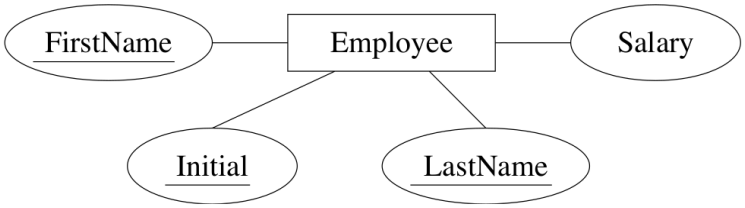
(2). binary relationship types;

(3). existence dependencies;

(4). general cardinality constraints.

2.2. (Primary Keys) A **primary key** is a selection of attributes for an entity set for which facts serve as the means of reference to its entities. Primary keys are labeled with *underlines*.

2.3. Example: `FirstName`, `Initial`, and `LastName` together serve as the primary key for `Employee`. It's important to note that a primary key may consists of more than one attributes.



2.4. (Binary Relationships) The idea of "one" is represented by an arrow.

- Many-to-Many (N:N): An entity in one entity set can be related to any number of entities in the other and the converse also holds.

$$\square - \diamondsuit - \square$$

- Many-to-One (N:1): Each entity in one entity set can be related to at most one entity in the other entity set, but no such limit exists for the converse.

$$\square \to \diamondsuit - \square$$

- One-to-Many (1:N): Inverse of many-to-one.

$$\square - \diamondsuit \leftarrow \square$$

- One-to-One (1:1): Each entity in one entity set can be related to at most one entity in the other, and the same holds for the converse.

$$\square \to \diamondsuit \leftarrow \square$$

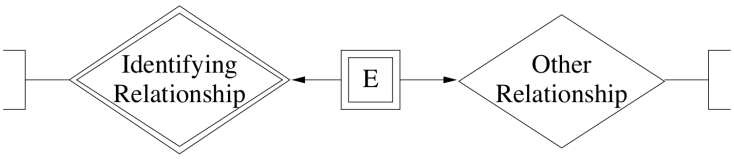
Note that none of these binary relationship types imply any *mandatory* participation of entities.

2.5. Sometimes, the existence of an entity depends on the existence of another entity. If x is **existence dependent** on y, we call y a **dominant entity** and x a **subordinary entity**. The subordinate entity are surrounded by *double boundaries* (rectangle inside rectangle).

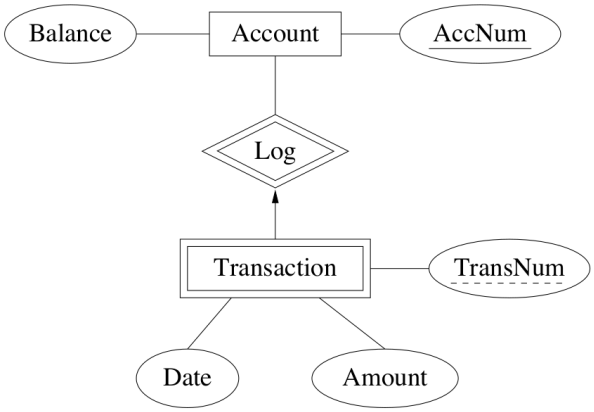
2.6. A **weak entity set** is an entity set containing subordinate entities. The set that a weak entity set is subordinate to is called the **identifying** or **owner entity set**. The relationship associating the weak entity set and the identifying entity set is the **identifying relationship**. Identifying relationships are labeled with *double boundaries*.

A **discriminator** is a selection of attributes for a weak entity set for which facts serve as the means of distinguishing subordinate entities for any given dominant entity. The discriminators (attributes of a weak entity set) are labeled via *dashed underlines*.

A weak entity set must be in a (N:1) relationship with at least one distinct entity set.



2.7. Example: Transactions are existence dependent on accounts. Moreover, this is a many-to-one relationship, i.e., each account may be associated with many transactions.



2.8. A **general cardinality constraint** determines lower and upper bounds on the number of relationships of a given relationship set in which a component entity must participate. An upper bound of N indicates that no upper bound exists. This is labeled via *component edge labelling*:

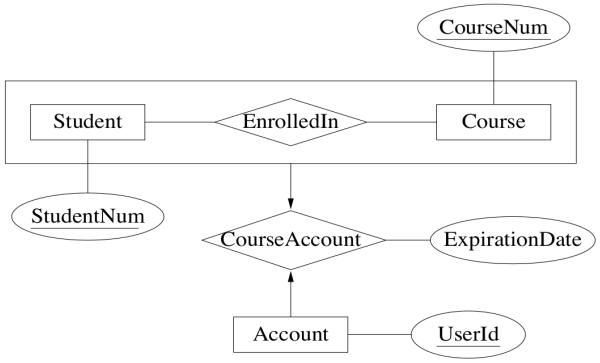
$$\square \underset{(l,u)}{--} \Diamond - - \cdots .$$

Section 3. Extensions to E-R Modeling

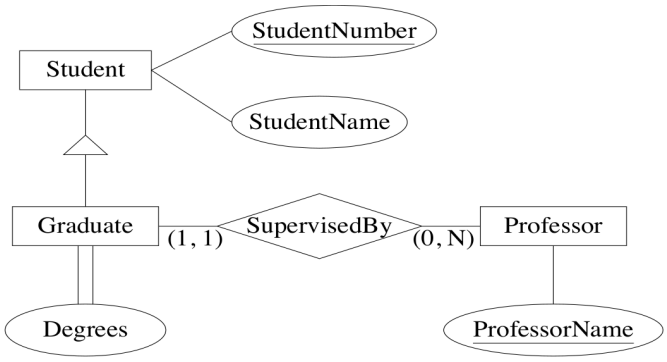
3.1. (Structured Attributes) Each attribute in ERM can be characterized by types:

- **Composite Attributes:** So far, the attributes are atomic or simple as they have no constituent parts. **Composite attributes**, on the other hand, can be divided into subparts. As an example, an `address` may be split into `street`, `city`, `province`, and `postal code`.
- **Multi-Valued Attributes:** A **multi-valued attribute** denotes a finite set of similar facts. For example, a person may have 0 to N phone numbers. When appropriate, upper and lower bounds may be placed on the number of values in a multi-valued attribute.
- **Derived attributes:** The value for this type of attributes can be derived from the value of other related attributes. For example, `age` may be derived from `dateOfBirth`.

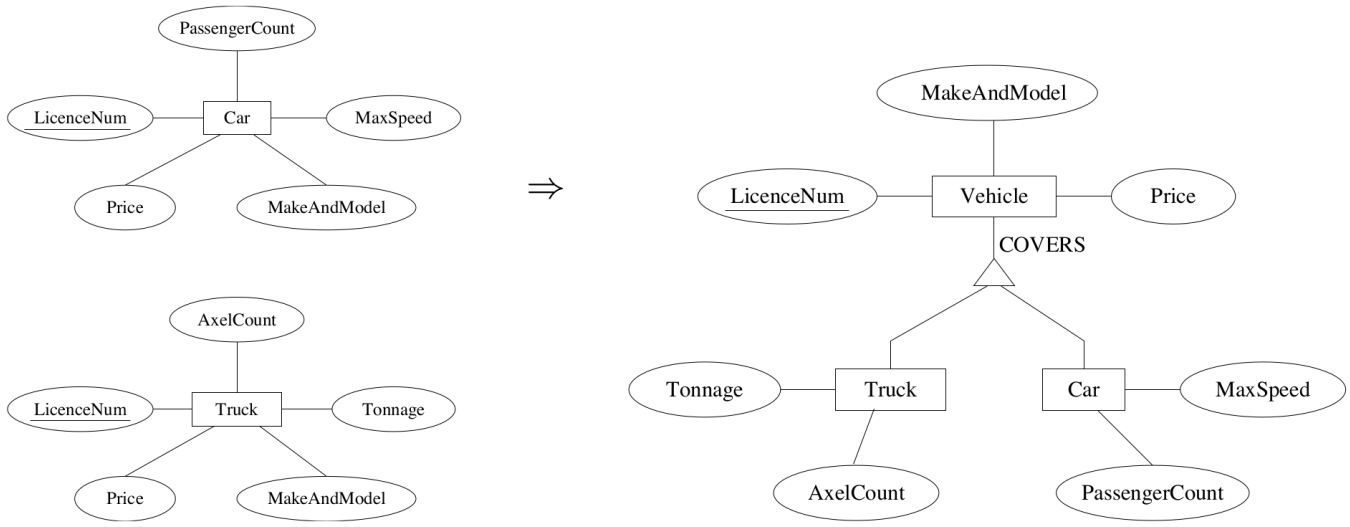
3.2. (Aggregation) A relationship set can be viewed as a higher-level entities. In the following example, each course account is allocated for a student enrolment (the big rectangle) in the course.



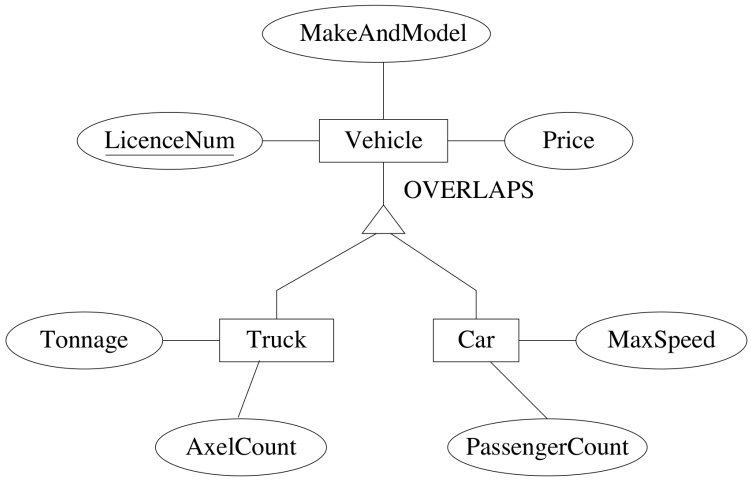
3.3. A **specialization** is an integrity constraint asserting that the entities of one entity set are also entities of another entity set. In the following example, the arrow indicates that graduate students are students who have a supervisor and a number of degrees (these two attributes are not possessed on general students).



3.4. (Generalization) A **generalization** is an integrity constraint asserting that entities of one entity ste are also entities of at least one of two or more other entity sets. A **total generalization** is one where each higher-level entity must belong to a lower-level entity set. Otherwise, the generalization is **partial**. The following example says a vehicle is also either a car or a truck.



3.5. (Disjointness) Two entity sets participating in a generalization are assumed to be **disjoint** by default. This can be overridden by a graphical annotation on the generalization. In the following example, the OVERLAPS indicates that there are entities that can be both a car and a truck, such as a utility vehicle.



Section 4. Design Methodology

4.1. An ER diagram for an information system is usually obtained from two sources:

(1). from parts of ER diagrams for existing information systems, or

(2). from informal requirements for the information systems obtained by *requirements elicitation*.

Issues emerge when authoring an ER diagram from informal requirements:

- attribute vs entity set;
- entity set vs relationship set;
- arity of relationship set;
- use of extended features;
- methodological considerations.

4.2. (Attributes vs Entity Sets) Rule of thumb:

- Is it a separate object?
- Do we maintain information about it?
- Can several of its kind belong to a single entity?
- Does it make sense to delete such an object?
- Can it be missing from some of the entity set's entities?
- Can it be shared by different entities?

An affirmative answer to any of the above suggests going with the entity set.

4.3. Remark: We can represent a relationship on n entity sets with n binary relationships.

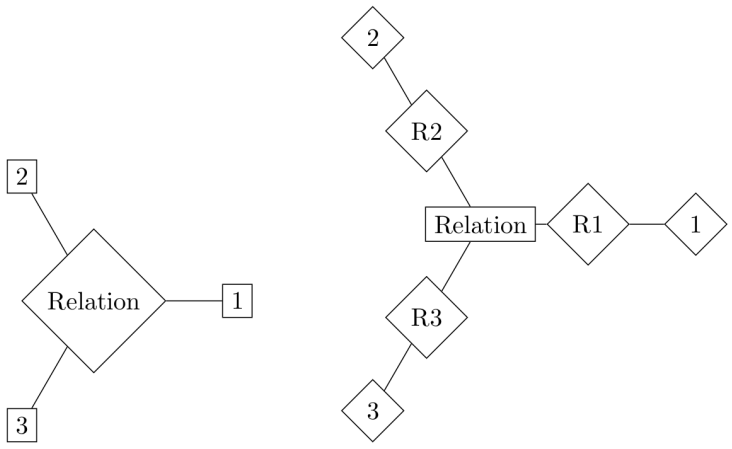


Figure 5.1: Credit: Ven

4.4. A simple methodology:

(1). Recognize entity sets.

(2). Recognize relationship sets and participating entity sets.

(3). Recognize attributes of entity and relationship sets.

(4). Define relationship types and existence dependencies.

(5). Define general cardinality constraints, keys and discriminators.

For each step, update the ER diagram and maintain a log of assumptions motivating the choices, and of restrictions imposed by the choices.

Section 5. ER Diagrams to Relational Schemata

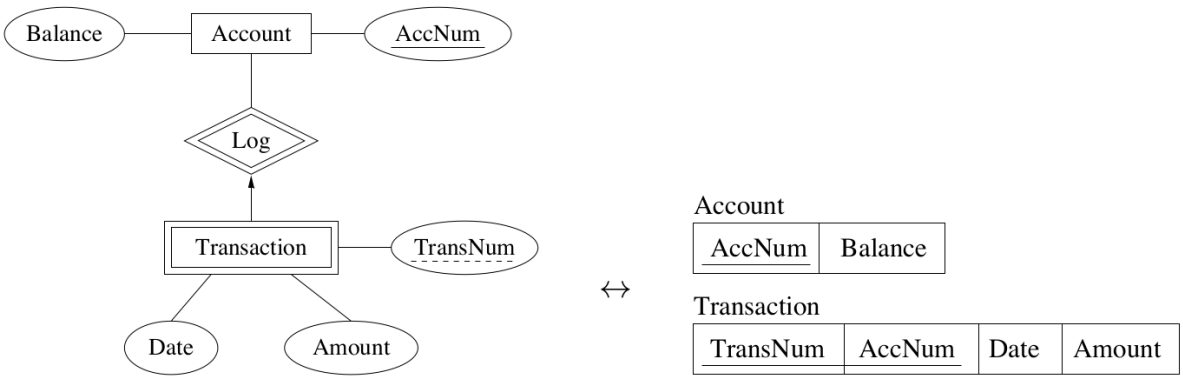
5.1. We now wish to obtain a logical design for a relational database from a conceptual design.

- Each entity set maps to a new table.
- Each attribute maps to a new table column.
- Each relationship maps to either new table columns for existing tables or a new table.

5.2. Each entity set maps to a new table and each attribute maps to a new table column.

- A *strong* entity set E with attributes $a_1, \ldots, a_n$ translates to a table E with attributes $a_1, \ldots, a_n$. Each entity maps to a row in the table and the primary key of the entity set maps to the primary key of the table.
- A *weak* entity set E with attributes $a_1, \ldots, a_n$ translates to a table E with attributes $a_1, \ldots, a_n, b_1, \ldots, b_m$, where b_i's are the primary key for the owner identity set. In other words, the columns of a table corresponding to a weak entity set should include:
  - attributes of the weak entity set.
  - attributes of the identifying relationship set, and
  - primary key attributes of entity set for dominating entities.

5.3. Example: In the design below, `Account` is a strong entity and `Transaction` is a weak entity. Translating `Account` is easy. For `Transaction`, we need its three attributes plus the primary key of its owner entity set, `AccNum`. Note that `TransNum` and `AccNum` together make up the primary key for `Transaction`.



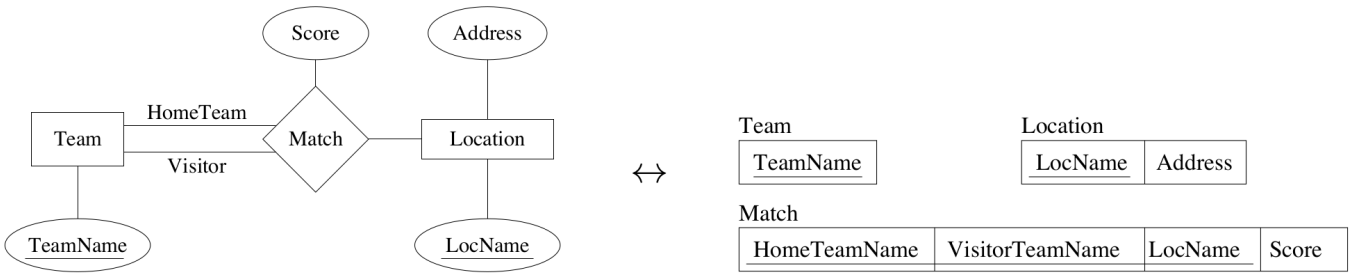
In addition, we have the following constraint on `Transaction`:

```
1  FOREIGN KEY (AccNum) REFERENCES Account;
```

5.4. Each relationship maps to either new table columns for existing tables or a new table.

- If the relationship set is an identifying relationship set for a weak entity, no action needed.
- If we can deduce the general cardinality constraint (1,1) for a component entity set E, then add the following columns to table E:
  - attributes of the relationship set, and
  - primary key attributes of remaining component entity sets.
- Otherwise, relationship set R translates to a new table R.
  - Let R denote the relationship set, $a_1, \ldots, a_m$ be the set of attributes formed by the union of primary keys of each entity sets participating in R, and $b_1, \ldots, b_n$ be the descriptive attributes. Then R is represented by the relation schema with columns
  $$\{a_1, \ldots, a_n\} \cup \{b_1, \ldots, b_m\}.$$
  - The primary key of table R is determined as follows:
    - If we can deduce the cardinality constraint $(0, 1)$ for a component entity set E, then take the primary key attributes for E.
    - Otherwise, the primary key is the union of primary key attributes of each component entity of relationship R.

5.5. Example: In the design below, `Team` and `Location` are strong entity sets and their translations are straightforward. We translate `Match` into its own table, with primary keys from its components: `TeamName` and `LocName`. Note that we combined role names and component entity set names to form more descriptive attribute names.



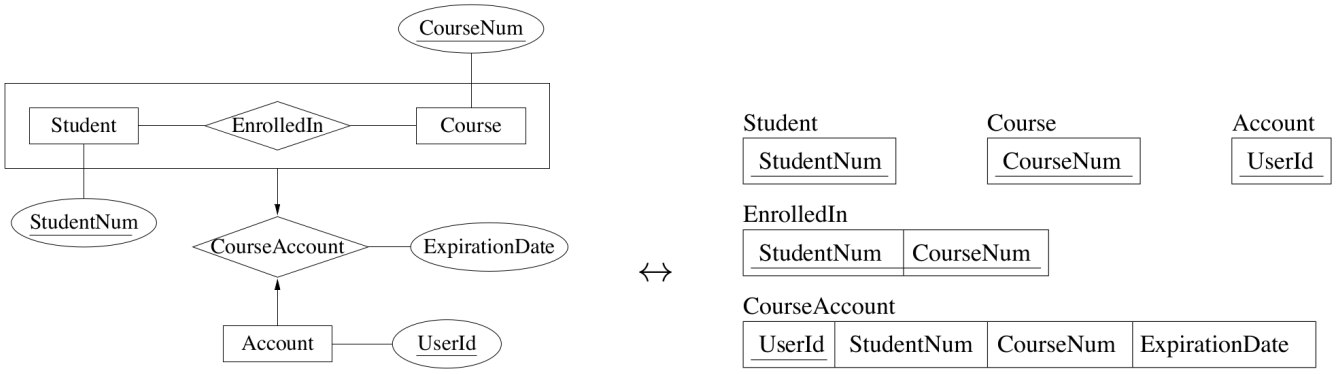
In addition, we have the following foreign key constraints on `Match`:

```
1  FOREIGN KEY ( HomeTeamName ) REFERENCES Team;
2  FOREIGN KEY ( VisitorTeamName ) REFERENCES Team;
3  FOREIGN KEY ( LocName ) REFERENCES Location;
```

Section 6. Mapping Extended Features

6.1. (Aggregation) Always map a relationship set R that is aggregated to a new table R. To represent a relationship involving the aggregation of R, treat the aggregation like an entity set whose primary key is the primary key of the table for R.

6.2. Example: Pay attention to what's in `EnrolledIn` and `CourseAccount`.

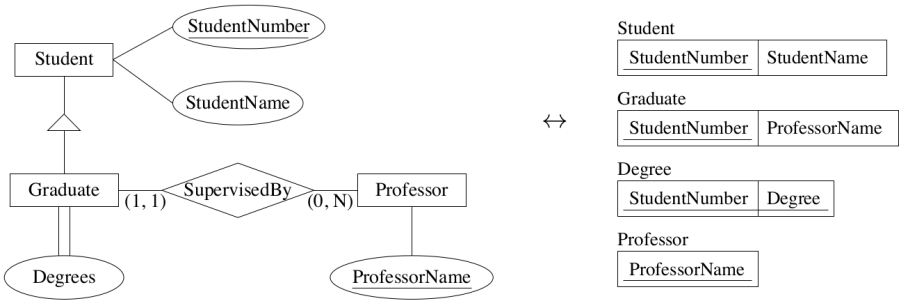


In addition, we have the following foreign key constraints:

```
1  -- In EnrolledIn:
2  FOREIGN KEY ( StudentNum ) REFERENCES Student;
3  FOREIGN KEY ( CourseNUm )  REFERENCES Course;
4  -- In CourseAccount:
5  FOREIGN KEY ( UserId ) REFERENCES Account;
6  FOREIGN KEY ( StudentNum, CourseNum ) REFERENCES EnrolledIn;
```

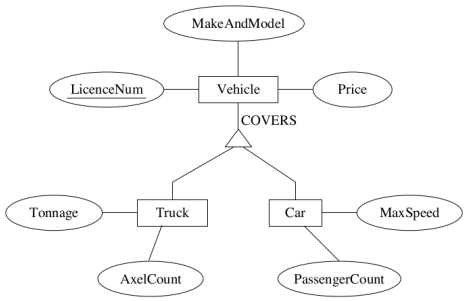
6.3. (Specialization) Treat an entity set which is a specialization of one or more other entity sets as a weak entity set with an empty discriminator set and that is existence dependent on each of the other entity sets.

6.4. Example: We omit the foreign key constraints.



6.5. (Generalization) Treat a generalization of n entity sets as n specializations and add additional constraints for converge and, if required, for disjointedness.

6.6. Example: We omit the foreign key constraints.



↔

Vehicle

LicenceNum	MakeAndModel	Price

Truck

LicenceNum	Tonnage	AxelCount

Car

LicenceNum	MaxSpeed	PassengerCount

6.7. (Using Views) Sometimes, an entity set can be mapped to a view instead of a table, which would increase efficiency and transparency. Here are two cases where this is allowed:

- An entity set E that is a specialization of one parent entity set and that satisfies the following conditions qualifies:
  (1). only typing attributes are declared on E,
  (2). no foreign key constraints reference E, and
  (3). all entity sets that are specializations of E are also mapped to views.

  Two more remarks:
  - Need to add a new two-valued typing attribute is $-E$ together with E's existing typing attributes to the parent entity set to enable a view definition for the mapping of E.
  - Multiple typing attributes can be replaced with a single typing attribute when underlying entity sets are disjoint.
- An entity set E that is a generalization of two or more other child entity sets and that has no foreign key constraints that reference E qualifies. Also need to ensure all attributes on E are defined on each child entity set to enable a view definition for the mapping of E.

6.8. Example: Using the same design as above, we have

Truck

LicenceNum	MakeAndModel	Price	Tonnage	AxelCount

Car

LicenceNum	MakeAndModel	Price	MaxSpeed	PassengerCount

```
1  CREATE VIEW Vehicle AS (
2      ( SELECT LicenceNUm, MakeAndModel, Price FROM Truck ) UNION
3      ( SELECT LicenseNum, MakeAndModel, Price FROM Car ))
```