COMP3311: Database Management Systems

Instructor: Dr. Fred LOCHOVSKY, HKUST

Notes by Joyce Shen

1 Database Management Systems

1.1 Basics and Definitions

Definition 1.1 (Database)**.** A **database** is a collection of related data (facts, i.e., age, salary, name, etc). Databases are designed and built for data with a specific purpose + inherent meaning.

Definition 1.2 (Database Management System (DBMS))**.** A **database management system DBMS** is a general purpose software system used to manage databases and allows for the defining, storing, manipulating, sharing, and protection of data.

Before DBMSs, **file systems** were used for data management – drawbacks included data duplication (inconsistency/atomicity of updates/concurrency issues, isolation of related data) constraint issues, security problems, etc. **DBMS**s addresses these issues by integrating + unifying an organization's data, supports concurrent manipulation, and controlls data definition *centrally.*

Definition 1.3 (Data Model)**.** A **data model** is the language used by a DBMS to describe how data is organized and accessed (state restrictions). May include:

- **data structure types** (logical organization)
- **integrity constraints** (restrictions on properties + relationships)
- **operations** (how data is accessed: RIUD – read, insert, update, delete).

Definition 1.4 (Database Schema)**.** A **database schema** a is a description of (some/all) the data in a database according to a data model – changes *infrequently.*

Definition 1.5 (Database Instance)**.** A **database instance** refers to the actual content of the database at a particular point in time, conforms to the schema – changes *frequently.*

An issue in application development is **dependence of code on data** – a database system provides **levels of abstraction**:

- **View level**: Defines a subset (part) of the database for a particular application using a view schema; can hide sensitive info or add information not actually stored (calculates age instead of storing it) – provides **logical data independence** + shields applications from changes in the logical database structure
- **Logical level**: Describes what data are stored in the database and the relationships among the data using a logical schema (how are students related to courses); hides details of how data is physically stored – provides **physical data independence** + shields applications from changes in the physical database structure
- **Physical Level**: Describes how the data are stored on disk using a physical schema (e.g., divide the student records into 3 partitions and store them on disks 1, 2 and 3.)

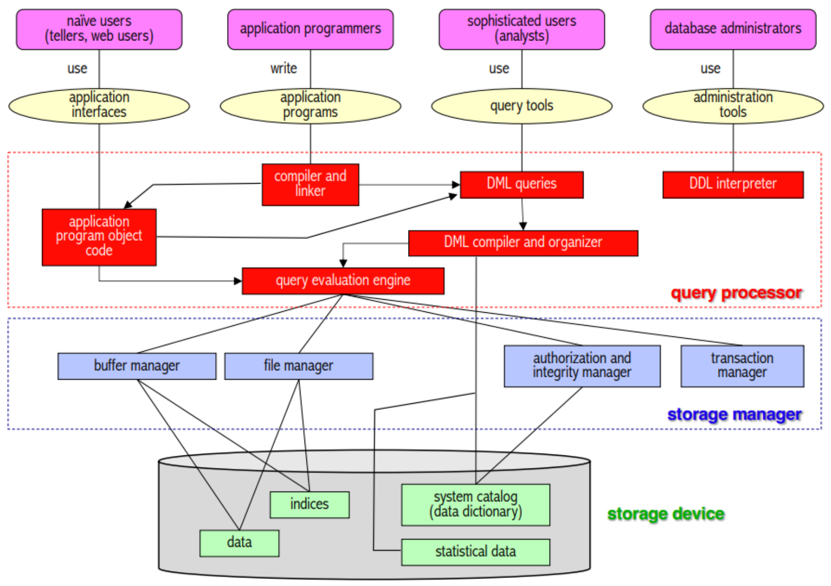
Relational DBMSs provide data independence via schemas following the three levels of data abstraction approach.

- The **semantics** (descriptions) are stored separately from the **data** (facts)
- All data must conform to the schema
- Applications are simpler to develop

Non-Relatioal (No-SQL) DBMSs generally do not use schemas.

- The **semantics** are either encoded in the application (code) or embedded with the **data**
- A lack of schema = flexibility w/ data representations
- Applications are more complex to develop
- Other benefits include scalability (horizontal scaling as opposed to vertical), availability (multiple servers instead of single), and cost (open-source software)

1.2 General DBMS Architecture



1.2.1 Storage Manager

The **storage manager** provides the interface between the raw data stored in the database (i.e., on physical storage devices) and the application programs/queries that access the database (responsible for storing, retrieving, and updating data). Typically includes buffer manager, file manager, authorization and integrity manager, and transaction manager.

1.2.2 Query Processor

The **query processor** translates application requests (i.e., queries and updates) into efficient device-level operations. Includes two query languages:

- **Data definition language (DDL)**: used to *define* the database schema according to a data model
- **Data manipulation language (DML)**: used to *access and manipulate* data as organized by a data model
  - **Procedural DML**: user specifies both what data is required and how to get the data; many non-relational db systems provide only a basic DML (e.g., get/put)
  - **Non-procedural (Declarative) DML**: user specifies only what data is required not how to get the data; all relational db systems provide **Structured Query Language (SQL)** and some non-relational dbs provide a DML similar to SQL

1.2.3 Users

End Users:

- **Naive users**: use existing applications through a GUI
- **Application programmers**: develop applications that interact with a DBMS through DML calls
- **Sophisticated users**: issue queries either directly using a DBMS query language or via tools such as data analysis software

Database Administrator (DBA):

- Coordinates all activities of the db system
  - Defines and maintains schemas, physical organization
  - Monitors database performance, (grants) access
- Must have a good understanding of an enterprise's information resources and needs

2 Database Design

Goals include: meet data content requirement of users, provide an intuitive structure, and do it efficiently (support data processing requirements and any performance objectives).

The design process involves:

1. **Requirement Analysis → Requirement Specification** – understands application domain + identifies data requirements
2. **Logical DB Design → Schema(s)** – describes how to represent the data requirements in a DB; for relational DBs, may include:
   - **Conceptual schema**: describes reqs using a DBMS-independent model (e.g., E-R data model)
   - **Logical schema**: describes DB using *data definition language (DDL)* of the target DBMS (e.g., SQL DDL)
3. **Physical DB Design → Physical Schema** – describes how the logical schema is stored on storage media

2.1 Entity-Relationship (E-R) Data Model

2.1.1 General Concepts: Entities, Attributes, and Relationships

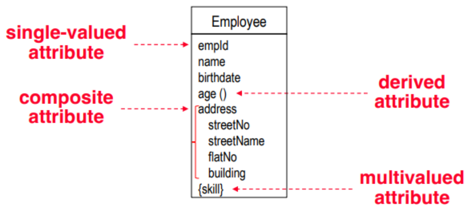
The **Entity-Relationship (E-R) Data Model** describes a DB's overall structure at the *logical level* (DBMS independent) Employs three basic concepts:

- **Entity**: something about which we want to keep data
- **Attribute**: properties of entities; what data to keep
- **Relationship**: among entities

Definition 2.1 (Entity (Type))**.** An **entity (type)** describes a set of *entity instances* with common properties, relationships, and semantics

An entity **type** is a description (e.g., Employee), an entity **instance** is the actual data (e.g., John Smith), and the entity **set** is the collection of all data of the same type (e.g., all employees)

Definition 2.2 (Attribute)**.** An **attribute** is a property of an entity + describes the data values of that property; Attribute have unique names (within the same entity) and are typically physically stored (think hardcoded), but can also be **derived** (calculated). Attributes can also be `null` (but try to avoid).



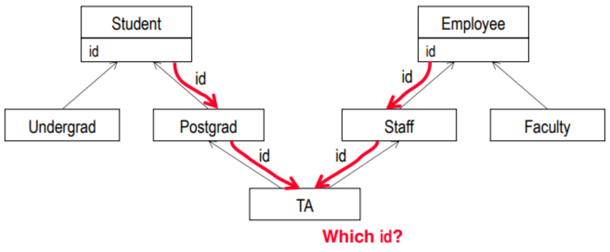
Entities can undergo **generalization/specification** – a relationship between the same kind of entities playing different roles (think inheritance in OOP)

- **Generalization**: bottom up
- **Specialization**: top down
- **Supertypes** are the parent type and **subtypes** are the children
- Can be **user-defined** (det. by schema designer and not value of attributes) or **attribute-designed** (det. by a predicate (i.e., logical condition) on an attribute e.g., Account has an attribute called *accountType*, with values {*Checking, Savings*})
  - A **discriminator**: is an attribute of enumeration type that indicates which property of an entity is being abstracted by a generalization/specification

Elaborating on **inheritance** (a subtype taking on properties of its supertype), note that we extract the *common* attributes + relationships between entities and associate them with the supertype, inheriting them to the subtype.

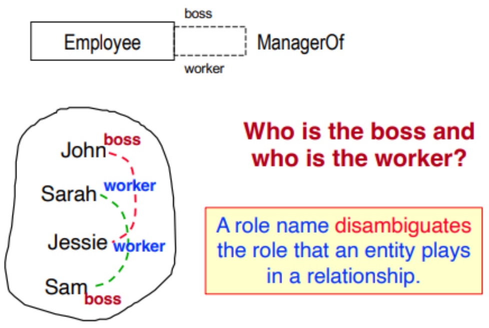
- We need only define each property of an entity in one place (reduces redundancy, promotes reusability, simplifies modification)
- Subtypes can add new properties (attributes, relationships)
- Inheritance should not exceed 2-3 levels

Note that **multiple inheritance** (1+ parent) may result in conflicts w/ attribute names – a property found from the same ancestor entity found along more than one path is *inherited only once*. Therefore attributes should be assigned meaningful and unambiguous names.



Definition 2.3 (Relationship)**.** A **relationship (type)** is a description of a set of relationships with common properties and semantics; conceptually, are bi-directional

The relationship **degree** is the number of entities that are involved (can be **unary/reflexive** – relating to itself, **binary** – relating two, or **ternary** – relating three (can often be expressed as two binary relationships, but not always!), higher degrees are possible, but very rare). Relationship **attributes** are variable-esque/parameter-esque values attached to relationships. A **role name** is assigned to *one side* of a relationship to identify the role an entity plays; necessary to use role names in unary relationships.


Employee
boss
worker
ManagerOf
John boss
Sarah worker
Jessie worker
Sam boss
Who is the boss and who is the worker?
A role name disambiguates the role that an entity plays in a relationship.

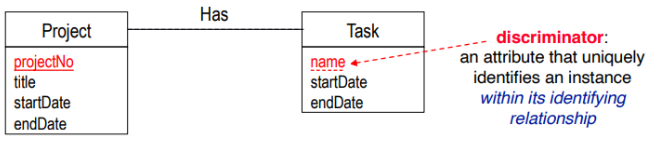
2.1.2 Constraints

Definition 2.4 (Constraint)**.** A **constraint** is a logical restriction/property of data s.t for any set of data values we can det if the constraint holds (with the expectation that it is always true) and which DBMS can enforce. Constraints add additional semantics (meaning) to data – to more accurately reflect application requirements.

A **domain constraint** restricts an attribute to have only certain values; can be specified as a *data type* or *predicate (logical condition)*

Keys:

- **Candidate key**: a minimal set of attributes that uniquely identifies an entity instance (may not be unique to an entity)
- **Primary key**: a specific candidate key selected by DB engineer – has enforcement implications: DBMS automatically enforces uniqueness, while uniqueness is not automatically enforced for other candidate keys
  - **Strong entities** have a primary key
  - **Weak entities** do not have a primary key and MUST be associated with an **identifying entity** (through an **identifying relationship** (with *total* participation) denoted by a solid line) to exist. A weak entity can be related to more than one identifying entity but may depend on only some of them for its existence (not all related entities need to be identifying)
    - A **discriminator** (if present), uniquely identifies a weak entity instance within its identifying relationship


Project
projectNo
title
startDate
endDate
Has
Task
name
startDate
endDate
discriminator:
an attribute that uniquely identifies an instance
within its identifying relationship

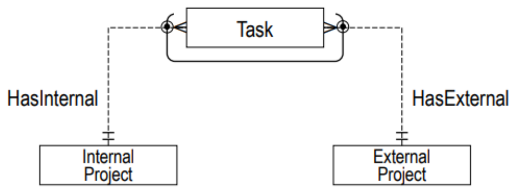
- Reminder that candidate + primary keys may be composed of a *set* of attributes – **composite** candidate/primary keys

Entity Generalization Constraints: **Coverage** is specified as one from disjointness and one from completeness.

- **Disjointness**: can be **overlapping** (supertype can relate to more than one subtype e.g., a Company can be both a Customer and a Supplier) or **disjoint** (supertype can relate to at most one subtype e.g., an Employee can be either a Secretary or an Engineer, not both)
- **Completeness**: can be **partial** (supertype does not need to relate to a subtype e.g., Loan doesn't need to be Mortgage, Tax, or Car) or **total** (supertype must relate to at least one subtype e.g., a Customer must be either a Personal or Business customer)

Relationship Constraints: specifies the number of relationship instances in which an entity instance may participate in a specified relationship type

- **Cardinality** the MAX number each entity E_1 *may* participate in R
- **Participation** the MIN number each entity E_1 *must* participate in R
- **Exclusion (XOR)** specifies that at most one entity instance, from among several entity types, can participate in a relationship with a single 'root' entity (e.g., a Task can be related to either an External or Internal project, not both)

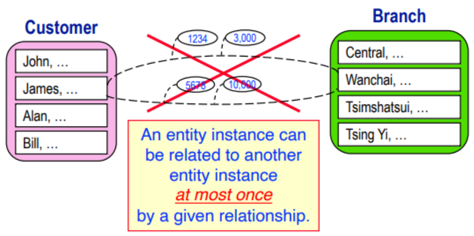


Relationships (cardinalities) may be

- **One-to-One (1:1)**: Employee and Office
- **One-to-Many (1:N)**: Employee and Department
- **Many-to-Many (N:M)**: Employee and Project

2.1.3 Design Choices

Placing an Attribute: relationship attributes may be needed if an attribute value exists only when a relationship exists
Strong vs Weak Entity: when the concept can be *uniquely identified* in the application domain (i.e., it has a key), it should be **strong**, otherwise it should be **weak**
Entity vs Relationship: when the concept represents *something distinct* in the application domain with *several properties*, it should be an **entity**; when the concept isn't distinct and/or has *no property of interest* it should be a **relationship** – Note that there can only be one relationship instance given relationship type between the same two entity instances



2.2 E-R to Relational Schema Reduction

2.2.1 The Relational Model

The **relational model** represents the data for an application as a collection of tables. Note that a table is not a relation, but a convenient way to look at a relation (and their instances, or tuples). This model uses only two concepts: **relation** and **attribute**.

A set of **relation schemas** define a **relational dabatase** and the data that can be stored in it. A **tuple** is an *ordered* sequence of n values $(v_1, v_2, ..., v_n)$ such that $v_i \in dom(A_i)$ or is `null`. The **cartesian product** over $dom(A_1), dom(A_2), ..., dom(A_n)$ is the set of all possible tuples $\{t_1, t_2, ..., t_m\}$ with $t_i \in dom(A_i)$. Note that the Cartesian product is not commutative, since domain order matters.

A **relation (instance)**: r over the relation schema R is

$$r(R) \subseteq dom(A_1) \times dom(A_2) \times ... \times dom(A_n)$$

Properties of Relations

- Tuples are unordered (they are *sets*)
- Attribute values are atomic (multivalued + composite attributes are not allowed in a relation even though they are allowed in the E-R model)
- **Degree** of a relation = number of attribute
- **Cardinality** of a relation = number of tuples (not the same as cardinality constraint in E-R model)

Keys

- **Superkey**: a superkey, S, of relation $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes $S \subseteq R$ s.t. for any two tuples $t_1, t_2 \in r(R), t_1[S] \neq t_2[S]$ – any set of attributes that can uniquely identify a tuple in $r(R)$ (does not have to be minimal)
- **Candidate key**: a minimal superkey
- **Primary key**: a specific candidate key selected by DB designer
- **Surrogate key**: a new attribute introduced into an entity to be the **primary key** for the entity; it normally has no meaning in the application domain – used to make **weak entities** $\rightarrow$ **strong** or to replace a **strong entity's key** if it consists of many attributes with a single attribute to facilitate schema reduction – usually invisible to the user and typically have sequentially assigned integers are values

Constraints:

- **Entity integrity constraint**: primary keys cannot contain `null` values
- **Referential integrity (foreign key) constraint**: given two relations S and T, T may *reference* relation S via a set of attributes fk_s that forms the primary key of S
  - The attributes fk_s in T are called a **foreign key** – the value of the foreign key in a tuple of T must either be equal to the value of the primary key k_s of a tuple S or be entirely `null`
    - If an entity in an E-R schema has **total** participation, **on delete cascade**
    - If an entity in a E-R schema has **partial** participation, **on delete set null**
  - A foreign key represents a 1:1 or 1:N relationship
  - Cardinality/participation constraints in an E-R schema reduce to referential integrity constraints in a relational schema

2.2.2 Reduction

Reducing Strong Entities
For a strong entity S, create a relation R_s with all the attributes of entity S, *excluding* multivalued and composite attributes. The primary key for relation R_s is the primary key for entity S.

Reducing Generalizations/Specializations
Option I: Reduce *all entities* $\rightarrow$ relation schemas

- Create a relation schema for *each entity* (supertype and subtype)
- For each *subtype*'s entity, add:
  1. The **primary key** k of the *superkey* as a *foreign key* fk (which becomes the primary key)
  2. A **foreign key constraint**: foreign key fk references *supertype-relation*(k)
  3. A **referential integrity action**: on delete cascade

Option II: Reduce only *subtype entities* to relation schemas

- Crease a relation schema for each *subtype entity*
- For each relation schema, add all the attributes of the supertype entity (the primary key is the primary key for the supertype entity)
- Should only be used for **total, disjoint generalizations/specializations**

Reducing Composite/Multivalued Attributes – If a *relationship* has a composite/multivalued attribute, convert it to an entity before reducing
For a **composite** attribute C in a strong entity S, either:

1. Create a single attribute by concatenating the composite attribute
2. Create a separate attribute for each component of the composite attribute

For a **multivalued** attribute M in a strong entity S:

- Create a relation schema SM with an attribute m that corresponds to the multivalued attribute $\{m\}$ and an attribute fk_s corresponding to the primary key for entity S
- The **primary key** for SM is the union of all its attributes and add:
    1. A **foreign key constraint**: foreign key fk_s references S(k)
    2. A **referential integrity action**: on delete cascade

Reduce Weak Entities – A weak entity T that depends on a strong entity S:

1. Create a relation schema R_T with attributes of T
2. Include attributes a_R of relationship R in relation R_T
3. Include as **foreign key attributes** fk_s in relation R_T the primary key attributes k_s of the strong entity S
4. The **primary key** for relation R_t is the *union of the foreign key attributes fk_s and the discriminator d_T (if any)* for the weak entity T and add:
    (a) A **foreign key constraint**: foreign key fk_s references S(k)
    (b) A **referential integrity action**: on delete cascade

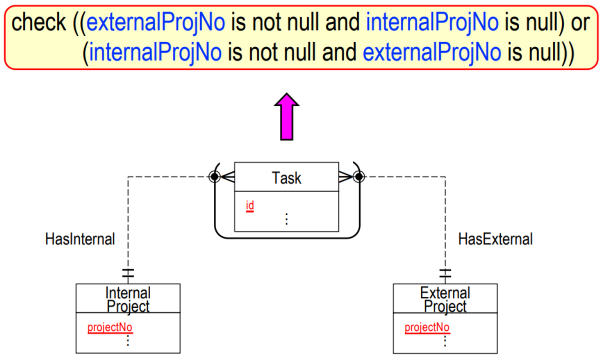
Reduce (binary) Relationships

1. Create a new relation schema R_R, include as **foreign key attributes** in R_R the primary keys of the entities related by relationship R and include attributes a_R of R (if any) as attributes of relation R_R
2. The **primary key** of relation R is:
    (a) **1:1** – primary key of either entity
    (b) **1:N** – primary key for eneity on N-side
    (c) **N:M** – union of both primary keys
3. Add a **foreign key constraint** for each foreign key with the **referential integrity action** on delete cascade

Relational Schema Combination
For **1:1** relationships, the relation schema for the relationship can be combined with the relation schema for either entity. For **1:N** relationships, the relation schema should be combined with **N-side** entity. Add:

1. A **foreign key constraint** for the foreign key
2. A **referential integrity action** for the foreign key that is det by the *participation* constraint of the entity into which the *foreign key is placed*
   - **partial**: on delete set null
   - **total**: on delete cascade

Reduce Exclusion Constraints
Option I: An additional constraint is needed to ensure that only one of entities has a value and the other is null – check clause



```
check ((externalProjNo is not null and internalProjNo is null) or
        (internalProjNo is not null and externalProjNo is null))
```

Option II: Add an attribute that identifies which entity an instance is – only valid if constrained entity types have keys of the same data type. Cannot specify referential constraints or actions for but the exclusion constraint is automatically enforced.

2.3 Relational Database Design

2.3.1 Functional Dependencies

Functional dependencies can be used to refine a relation schema reduced from an E-R schema by iteratively decomposing it into a certain **normal form** (ie. normalization).

Definition 2.5 (Functional Dependency)**.** Let R be a relation schema, X, Y be sets of attributes in R, and f be a time-varying function from X to Y. Then $f : X \mapsto Y$ is a functional dependency (FD) if, at every point in time, for a given value of x in X there is ever at most one value of y in Y.

In other words, a functional dependency satisfies the **unique mapping** property of a function. $\forall x \in X, \exists y_1, y_2 \in Y \;.\; x = y_1, x = y_2, \implies y_1 = y_2$. With a functional dependency $f : X \mapsto Y$, X is called the **determinant set** or LHS of the FD while Y is the **dependent set** or RHS.

A functional dependency $X \mapsto Y$ is **trivial** if $Y \subseteq X$ (aka Y appears on both the LHS and RHS of the FD). Trivial FDs hold for all relation instances. A FD is **nontrivial** if $Y \cap X = \emptyset$. Nontrivial FDs represent constraints and limit the set of legal relations.

An FD is a generalization of a **key**. For example, the relation `PGStudent(studentId, name, supervisor, specialization)` can be written as `studentId` $\rightarrow$ `name, supervisor, specialization` becaues the key (studentId) determines the values of all the attributes.

Given a set of functional dependencies F, there are (potentially unlimited) other FDs logically implied by F, which can be found by using **inference rules**:

1. **Reflexivity**: if $Y \subseteq X$, then $X \mapsto Y$
2. **Augmentation**: $X \mapsto Y \vDash XZ \mapsto YZ$
3. **Transitivity**: $X \mapsto Y$, $Y \mapsto X \vDash X \mapsto Z$
4. **Union**: $X \mapsto Y$, $X \mapsto Z \vDash X \mapsto YZ$
5. **Decomposition**: $X \mapsto YZ \vDash X \mapsto Y$ and $X \mapsto Z$
6. **Pseudotransitivity**: $X \mapsto Y, WY \mapsto Z \vDash WX \mapsto Z$

The first three rules form **Armstrong's Axioms**, from which the last three rules can be derived. Armstrong's Axioms are **sound** (all FDs inferred are valid) and **complete** (there are no other FDs that are true besides what can be inferred). Thus, the set of all FDs logically implied by F is called the **closure** of F, denoted F^+.

Definition 2.6 (**Attribute** Closure)**.** The closure of X under F (denoted X^+) is the set of **attributes** that are functionally determined by X under F.

$$X \mapsto Y \text{ is in } F^+ \iff Y \subseteq X^+$$

Attribute closures can be used to test for superkeys. If X^+ contains all attributes of R, then X is a superkey. If X is also **minimal** (all attributes in the key are essential), then it is a **candidate key**. An attribute that is part of *any* candidate key is a **prime attribute** (otherwise, it is **nonprime**). Attribute closures can also test functional dependencies and compute the closure of F.

Sets of functional dependencies may be *redundant* (containing FDs that can be inferred from the others or containing FDs with **extraneous/redundant attributes**). To remedy redundancy, look at the **canonical cover** of F.

Definition 2.7 (Canonical Cover)**.** A canonical cover of F, denoted F_c, is a set of functional dependencies such that

- F and F_c are equivalent (i.e., have the same closure F^+)
- F_c contains no redundancy
- The LHS of each FD in F_c is unique

Every set of FDs has a canonical cover, but it is not necessarily unique. Testing for FD violations with F_c is usually simpler than with F. The algorithm to construct a canonical cover is as follows:

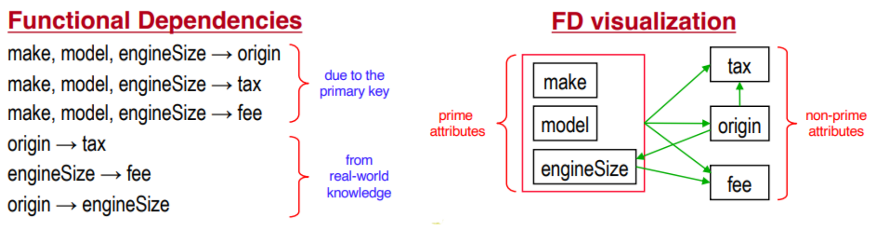
1. $F_c = F$
2. Use the **union rule** to replace any FDs in F_c with the same LHS ($X \mapsto Y, X \mapsto Z$ becomes $X \mapsto YZ$)
3. Delete **extraneous attributes** from FDs in F_c
4. Repeat steps 2 and 3 until F_c does not change

2.3.2 Normalization

Normalization uses functional dependencies to decompose (i.e., break up) an unsatisfactory relation schema into fragments (two or more relation schemas) that possess more desirable properties. Normalization is expressed in terms of **normal forms**, but normal forms themselves *do not guarantee* good design.

There are a few main goals/guidelines for normalization:

1. **Clear semantics** for attributes: each relation schema should have an unambiguous meaning; should not be mixing attributes from multiple entities into a single relation schema
2. **Minimize null** values, as they lead to interpretation and operation execution problems (aggregation operations)
3. **Minimize redundancy** so no **operation anomalies** (insertion, deletion, or updating) occur in the relation instances. A relation schema has redundancy if there is a FS where the LHS is *not* a primary key.
4. **Lossless decomposition**: a decomposition is lossless if the original relation instance can be recovered from the decomposed relation fragments. A decomposition of R into R_1 and R_2 is *lossless* iff at least one of $R_1 \cap R_2 \mapsto R_1$ or $R_1 \cap R_2 \mapsto R_2$ is in F^+ – a.k.a. the *common attributes* of R_1 and R_2 must be a *superkey* of R_1 or R_2
5. **Preserve functional dependencies**: if a FD doesn't appear in any relation schema (and is "lost"), the constraint may be much more difficult to enforce (might require computing joins, which is expensive)



Functional Dependencies

make, model, engineSize → origin
make, model, engineSize → tax
make, model, engineSize → fee
due to the primary key

origin → tax
engineSize → fee
origin → engineSize
from real-world knowledge

First Normal Form (1NF) - a relation schema is in first normal form if all its attributes are **atomic** (no multi-valued/composite attributes); relation schemas are always in 1NF through the method of reducing an E-R schema to a relation schema

Second Normal Form (2NF) - if all **non-prime** attributes are *fully* functionally dependent on every candidate key; in 2NF iff for each FD $X \rightarrow A \in F^+$:

1. $A \in X$ (FD is trivial) OR
2. X is **not** a proper subset of a candidate key for R OR
3. A is a **prime attribute** for R

Alternatively, R is *not* in 2NF if for *any* FD $X \rightarrow A$, the FD is *nontrivial*, X *is* a proper subset of a candidate key for R, and A contains no part of a candidate key – i.e., a subset of a candidate key cannot determine any part of another candidate key.

Third Normal Form (3NF) - every **non-prime** attribute is non-transitively dependent on every candidate key; in 3NF iff for each FD $X \rightarrow A \in F^+$:

1. $A \in X$ (FD is trivial) OR
2. X is a **superkey** for R OR
3. A is a **prime attribute** for R

Alternatively, R is *not* in 3NF if for *any* FD $X \rightarrow A$, the FD is *nontrivial*, X is *not* a superkey, and A is *not* a prime attribute.

The 3NF Decomposition Algorithm (3NF Synthesis Algorithm): – always **lossless join, dependency preserving** – Let R be the initial relation schema with FDs F and S be the set of relation schemas (initially $\emptyset$):

1. Compute a **canonical cover** F_c of F
2. For each FD $X \rightarrow Y$ in F_c, create a relation schema and add it to S: $S = S \cup (X, Y)$
3. If no schema contains a *candidate key* for R, choose any candidate key k and add it to S: $S = S \cup K$

Boyce-Codd Normal Form - every determinant (LHS) of its FDs is a superkey; BCNF relations have no redundancy (that can be removed by FDs) and guarantees 3NF (and below). R is in BCNF iff: for each FD $X \to A \in F^+$:

1. $A \in X$ (FD is trivial) OR

2. X is a superkey for R

The BCNF Decomposition Algorithm: (a relation may *not* have a dependency preserving BCNF decomposition – but always lossless) – Let R be the initial relation schema with FDs F and S be the set of relation schemas (initially $\emptyset$):

1. Computer the **closure** F^+

2. Set $S = R$

3. Until all relations schemas in S are in BCNF:
   ```
   For each R in S
       For each FD X→Y that violates BCNF for R
           S = (S – R) ∪ (R – Y) ∪ (X, Y)
   ```
   Essentially, remove R from S, add a schema w/ same attributes as R but without Y, and add a second schema w/ attributes of X and Y (i.e., create a relation schema for $X \to Y$)

3 Relational Algebra (RA)

There are two mathematical query languages that form the basis for ”real” relational query languages (e.g., SQL) and for implementation:

- Relational algebra: procedural (step-by-step); need to describe how to compute a query result (focus in this course)

- Relational calculus: non-procedural (declarative); only need to state what query result is wanted

Definition 3.1 (Relational Algebra)**. Relational algebra** is an algebra where

- **operands** are either relations or variables that represent relations

- **operations** perform common/basic manipulations of relations

Relational algebra has a **closure property**, where each operation manipulates one or more relations and returns a relation as a result, allowing operations to be composed – (think functional programming in which functions are values.)

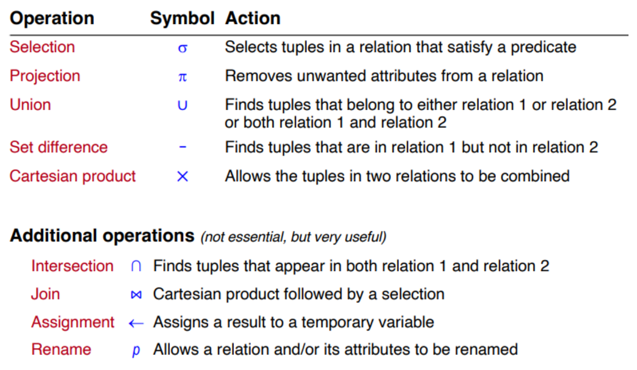
Relational algebra is very useful for representing + optimizing query exection plans (SQL optimizes) and is key to understanding how SQL is processed.

3.1 Basic Operations

A relational algebra expression is evaluated from the *inside out*.

- **Selection** $\sigma_C(R)$: selects tuples according to condition C returning a relation w/ a schema identical to the input
    - C has the form: `term` *`op`* `term` where `term` is an attribute name or constant and *`op`* is a comparison operator (e.g., $=, \neq, \leq, ...$)
- **Projection** $\pi_L(R)$: keeps only attributes in a projection list L (and removes duplicates)
- **Set Operators** $\cup, -, \cap, \times$: can be used between two (union compatible – same number of attributes + matching types) input relations
- **Join** $\bowtie$: a cartesian product ($\times$) followed by a selection: $R_1 \bowtie_C R_2 = R_1 \text{ JOIN}_C\ R_2 = \sigma_C(R_1 \times R_2)$
    - **θ-Join**: allows arbitrary conditions in the selection
    - **Natural Join**: combines two relations based on attributes with the same names (cannot qualify common join attributes) – project results on only one set of the common attributes
    - **Outer Join**: computes natural join + fills tuples that don't have a matching value for the common join attribute with `null`
        - All comparisons involving `null` are false (`null = null`)
        - Includes **left**, **right**, and **full** outer join – specifies which relation(s) to include all values of
- **Renaming** ρ: assigns a name to a relational algebra expression; necessary when taking a cartesian product or joining a relation with itself
    - $\rho_x(E)$ assigns name x to the result of E (E can be a relation name or a relational algebra expression)
    - $\rho_{x(A_1,A_2,...,A_n)}(E)$ assigns name x to the result of E and renames the attributes of E as $A_1, A_2, ..., A_n$

Summary:



Operation	Symbol	Action
Selection	σ	Selects tuples in a relation that satisfy a predicate
Projection	π	Removes unwanted attributes from a relation
Union	$\cup$	Finds tuples that belong to either relation 1 or relation 2 or both relation 1 and relation 2
Set difference	$-$	Finds tuples that are in relation 1 but not in relation 2
Cartesian product	$\times$	Allows the tuples in two relations to be combined

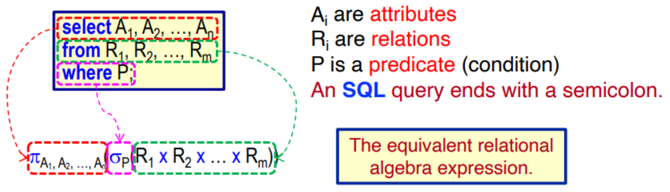
Additional operations *(not essential, but very useful)*

Intersection	$\cap$	Finds tuples that appear in both relation 1 and relation 2
Join	$\bowtie$	Cartesian product followed by a selection
Assignment	$\leftarrow$	Assigns a result to a temporary variable
Rename	ρ	Allows a relation and/or its attributes to be renamed

4 Structured Query Language (SQL)

4.1 SQL DML (Data Manipulation Language)

SQL DML, which is used in all commercial relational DBMSs, is based on **set** and **relational algebra** operations with certain modifications and enhancements. A SQL query follows the basic form


select $A_1, A_2, \ldots, A_n$
from $R_1, R_2, \ldots, R_m$
where P
$\pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$
A_i are attributes
R_i are relations
P is a predicate (condition)
An SQL query ends with a semicolon.
The equivalent relational algebra expression.

and returns a **relation** (which may contain duplicates). Queries can be composed.

4.1.1 Basic Structure and Operations

Projection:
`SELECT` clause: specifies attributes to include in query result (never actually changes the DB)

- Astricks `*` in `SELECT` clause denotes "all attributes"
- Attributes specified in `SELECT` clause must be defined in the `FROM` clause
- SQL doesn't remove duplicates by default, use keyword `DISTINCT` (use `ALL` to keep duplicates)
- Can contain arithmetic operations that operate on constants or attributes (DB is not changed, only query result)

Selection:
`WHERE` clause: specifies the criteria that tuples in the relations in the `FROM` clause must satisfy.

- Attributes specified in `WHERE` clause must be defined in the `FROM` clause
- SQL string values must be in single quotes, numeric values do not require quotes (and idiomatically are typically omitted)
- Comparison, boolean, and arithmetic operators/expressions can be used
  - `AND` is higher precedence than `OR`, so use parentheses accordingly
  - The keyword `BETWEEN` can be used (and `NOT BETWEEN`)
  - `null` is an exception – must use the `IS` keyword e.g., `WHERE district IS null`

Cartesian Product:
`FROM` clause: corresponds to relational algebra's cartesian product ($\times$)

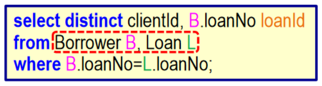
- Can be expressed with a comma: `FROM Borrower, Loan` or `CROSS JOIN` keyword: `FROM Borrower CROSS JOIN Loan`
- A `FROM` with more than one relation is rarely used without a join condition:
  - A natural join is done w/ `NATURAL JOIN` (in the `FROM` clause) – will automatically join on *all* the common attributes; use `USING` to specify which attribute(s); note that `NATURAL` is not allowed with `USING`
  - A θ-join can be specified by adding a join condition in the `FROM` or `WHERE` clause
  - An outer join is specified w/ `[NATURAL] {FULL|LEFT|RIGHT} OUTER JOIN` in the `FROM` clause
  - The join condition can also be specified in an `ON` clause; can be useful for explicitly specifying which attributes to join on

Set Operations:
`UNION`,`INTERSECT`, and `EXCEPT(MINUS)` operate on relations and correspond to relational algebra's $\cup$, $\cap$, $-$

- Each operation *removes duplicates* – to keep, use `ALL` keyword

Rename Attributes:
`AS` clause: `OLD_NAME AS NEW_NAME`

- The keyword `AS` is optional when used in the `SELECT` clause
- Can rename *relations* as well (**table alias, correlation variable, tuple variable**)
  - Useful for reusing long relations within a query
  - Required when comparing tuples in the same relation (i.e., a self join)
  - Omit `AS` in `FROM` clause

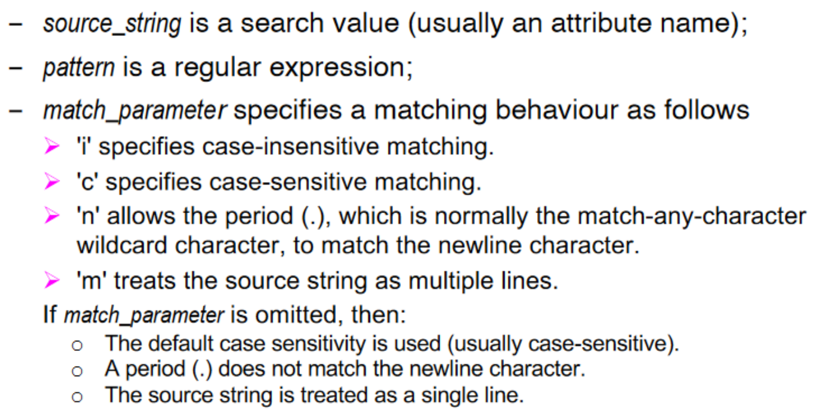


```
select distinct clientId, B.loanNo loanId
from Borrower B, Loan L
where B.loanNo=L.loanNo;
```

String Patterm Matching
`LIKE` operator is used to match characters in string values:

- `%` = any substring
- `_` = any single character
- \ = escape character, `''` to include single quotes

`REGEXP_LIKE` operator matches using **regular expressions**:
`REGEXP_LIKE(SOURCE_STRING, PATTERN, [MATCH_PARAMETER])`



- *source_string* is a search value (usually an attribute name);
- *pattern* is a regular expression;
- *match_parameter* specifies a matching behaviour as follows
  - 'i' specifies case-insensitive matching.
  - 'c' specifies case-sensitive matching.
  - 'n' allows the period (.), which is normally the match-any-character wildcard character, to match the newline character.
  - 'm' treats the source string as multiple lines.

  If *match_parameter* is omitted, then:
  - The default case sensitivity is used (usually case-sensitive).
  - A period (.) does not match the newline character.
  - The source string is treated as a single line.

Ordering Tuple Results
`ORDER BY` clause, with `ASC` and `DESC` options

- By default, `null` is the *largest* value, making it first in `DESC` and last in `ASC`, which can be changed with `NULLS LAST` or `NULLS FIRST`

Limiting Result Tuples
`FETCH` clause limits retrieval to only a portion of the rows

- E.g., `FETCH FIRST n ROWS ONLY`
- Can specify `WITH TIES` and discard rows with `OFFSET n ROWS`

4.1.2 Aggregate Functions

Definition 4.1 (Aggregate Functions)**. Aggregate functions** operate on a *relation attribute* and returns a relation with *one row and one column* (i.e., a single value)

- For `AVG, STDEV, SUM`, the input must be numbers
- For other aggregate functions, the input can be non-numeric (e.g., strings)
- All aggregate functions except `COUNT(*)` *ignore null values* and return a value of `null` for an empty collection – the count of an empty collection is 0

`COUNT(*)` variations:

- `COUNT(*)` = all tuples
- `COUNT(branchName)` = counts branchName

- `COUNT(DISTINCT branchName)` = counts distinct branchNames (shocker)
- `COUNT(DISTINCT *)` = illegal

`GROUP BY` – permits aggregate results to be displayed for groups; `GROUP BY x` returns a result for every `x`

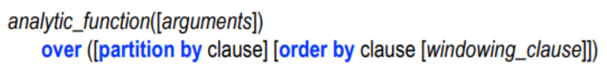
- Aggregate queries without a `GROUP BY` return a *single* value while queries with a `GROUP BY` return a value *for every group*
- Note that attributes in the `SELECT` clause must appear in the `GROUP BY` clause, but the opposite is not true
- `GROUP BY` clauses (and aggregate functions) are applied to join results when

4.1.3 Nested Subqueries and Set Operations

4.1.4 Analytic Functions

Definition 4.2 (Analytic Functions)**. Analytic functions** compute an *aggregate value* for the query result, possibly based on a grouping of rows of the query result called a *partition*.

Unlike **aggregate** functions, analytic functions can return multiple rows for each group. Analytic functions are the last set of operations performed in a query aside from the `ORDER BY` clause and can only appear in `SELECT` or `ORDER BY`.



```
analytic_function([arguments])
    over ([partition by clause] [order by clause [windowing_clause]])
```

- **analytic_function**: `SUM`, `RANK`, `AVG`, etc
- `PARTITION BY` clause: (optional), partitions the query result into groups based on one or more criteria (similar to a `GROUP BY` clause in a `SELECT`). If this clause is omitted, then the function treats all tuples of the query result as a single group.
- `ORDER BY` clause: (optional), specifies how data is ordered within a partition (similar to the `ORDER BY` clause in a `SELECT`)

4.1.5 Database Modification

4.1.6 SQL in Programming Languages

4.2 SQL DDL