

CS2102 — DATABASE SYSTEMS

PROF. CHAN CHEE YONG

PROF. XIAO XIAOKUI

DR. JEFFRY HARTANTO

*arsatis**

CONTENTS

1	Introduction	4
1.1	Integrity Constraints (ICs)	4
1.1.1	Key Constraints	5
1.1.2	Foreign Key Constraints	5
1.2	Data Abstraction & Independence	6
1.2.1	Transactions	6
2	Relational Algebra	7
2.1	Unary Operators	8
2.1.1	Selection	9
2.1.2	Projection	9
2.1.3	Renaming	10
2.2	Binary Operators	10
2.2.1	Set Operators	10
2.2.2	Cross-Product	10
2.3	Join Operators	11
2.3.1	Inner Join	11
2.3.2	Natural Join	11
2.3.3	Outer Join	11

*author: <https://github.com/arsatis>

3	SQL	12
3.1	Basics	12
3.1.1	Data Types	12
3.1.2	Tables and Comments	13
3.1.3	Comparison Predicates	13
3.2	Constraints	14
3.2.1	Assertions	16
3.3	Database Modifications	16
3.4	Queries	17
3.4.1	Subqueries	17
3.4.2	Scalar Subqueries	18
3.4.3	Additional Query Clauses	19
3.4.4	Aggregate Functions	19
3.4.5	GROUP BY clause	19
3.4.6	HAVING clause	20
3.4.7	Conceptual Evaluation of Queries	20
3.5	Advanced Features	21
3.5.1	Common Table Expressions (CTEs)	21
3.5.2	Views	21
3.5.3	Conditional Expressions	22
3.5.4	Pattern Matching	22
4	Conceptual Database Design	23
4.1	Entity-Relationship (ER) Model	23
4.1.1	Relationship Constraints	24
4.1.2	Weak Entity Sets	25
4.1.3	Aggregation	25
4.1.4	ISA Hierarchies	26
5	Procedural SQL	27
5.1	Host Language + SQL	27
5.2	PL/pgSQL	28
5.2.1	Variables	29
5.2.2	Control Structure	30
5.2.3	Cursors	31
5.3	SQL Injection	32
6	Triggers	33
6.1	Trigger Timing	34
6.2	Trigger Levels	34
6.3	Trigger Conditions	35
6.4	Deferred Triggers	35
6.5	Multiple Triggers	36
7	Normal Forms (I)	36
7.1	Functional Dependencies (FD)	37

7.1.1	FD Reasoning	37
7.1.2	Closure	37
7.2	Keys, Superkeys, and Prime Attributes	38
8	Normal Forms (II)	38
8.1	Non-Trivial and Decomposed FD	39
8.2	Boyce-Codd Normal Form (BCNF)	39
8.2.1	BCNF Decomposition	40
8.3	Third Normal Form (3NF)	41
8.3.1	3NF Decomposition	41

Note that lecture numbers 4, 5, and 6 are swapped, due to some constraints with the \LaTeX package used.

1 INTRODUCTION

Lecture 1
10th January 2022

data model: collection of concepts for describing data

A **database management system (DBMS)** is a software for managing large persistent data. It allows users to define and query data in terms of a **data model**. Some of its advantages include:

- data independence,
- efficient data access,
- data integrity and security,
- data administration,
- transaction management (concurrent access & crash recovery), and
- it being a query language.

schema: description of the structure of a database using a data model

A **relation schema** $R(A_1, A_2, \dots, A_n)$ specifies **attributes** A_1, \dots, A_n and **data constraints**. Each **instance** of schema R is a **relation**, which is a subset of

$$\{(a_1, a_2, \dots, a_n) | a_i \in \text{domain}(A_i) \cup \{\text{null}\}\}$$

schema instance: content of the database at a particular time

Each row in a relation is called a **tuple/record**; it has one **component** for each attribute of the relation.

Students			
<i>studentId</i>	<i>name</i>	<i>birthDate</i>	<i>cap</i>
3118	Alice	1999-12-25	3.8
1423	Bob	2000-05-27	4.3
5609	Carol	1999-06-11	4.0

Figure 1: Example of a relation.

domain: set of atomic values, e.g., integer, text, double

null: special value used to indicate that the value is either not applicable or unknown

The **relational data model** is a data model where data is modelled using relations (i.e., tables with rows & columns). **Cardinality** refers to the number of rows in the table, whereas **degree/arity** refers to the number of columns in the table. A **(relational) database** is a collection of tables/relations, and a **(relational) database schema** consists of a set of relation schemas.

1.1 Integrity Constraints (ICs)

data constraints: rules enforced on the data columns of a table that are used to limit the type of data that can go into a table, e.g., (*studentId*: int, *name*: string)

An **integrity/data constraint** is a condition that restricts the data that can be stored in a database instance. It is specified when a schema is defined, and checked when relations are updated. A **legal relation instance** is a relation that satisfies all specified ICs. DBMS enforces ICs, allowing only legal instances to be stored. Some types of integrity constraints include:

- **domain constraints**, which restrict attribute values of relations,
- **key constraints**, and
- **foreign key constraints**.

1.1.1 Key Constraints

A **superkey** is a subset of attributes in a relation that *uniquely identifies* its tuples. For example, *studentId* and $\{studentId, name\}$ would be valid superkeys for the following relation:

Students(*studentId*, *name*, *birthDate*, *email*)

A **key** is a *minimal subset* of attributes in a relation that *uniquely identifies* its tuples. In the example above, *studentId* is also a valid key, but the attribute pair $\{studentId, name\}$ is not a valid key.

In other words, key constraints include the following:

1. no two distinct tuples of a relation will have the same values in all attributes of the key.
2. no proper subset of the key is a superkey.
3. attribute values cannot be null.

A relation could have multiple keys, called **candidate keys**. One of the candidate keys will be selected to be the **primary key**. Referring to the previous example, *studentId* and *email* are candidate keys of the relation Students, and one of them could be selected as the primary key.

1.1.2 Foreign Key Constraints

A subset of attributes in a relation is a **foreign key** if it refers to the primary key of a second relation (refer to Figure 2 for an example).

Foreign key constraints require that:

- each foreign key value in a referencing relation must either:
 1. appear as a *primary key value* in the referenced relation, or
 2. be a null value.
- note that in SQL, foreign keys can reference columns which are unique (need not be a candidate or primary key)

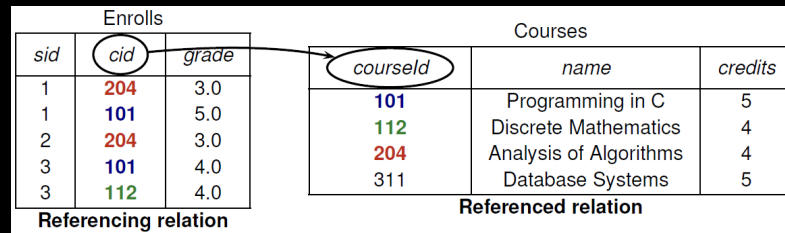


Figure 2: In this example, *cid* is a foreign key in *Enrolls* that refers to the primary key *courseId* in *Courses*.

1.2 Data Abstraction & Independence

Data in DBMS is described at 3 levels of abstraction:

1. **Logical schema:** the logical structure of data in DBMS.
2. **Physical schema:** how the data described by logical schema is physically organized in the DBMS.
3. **External schema:** a customized view of the logical schema for a specific group of users.

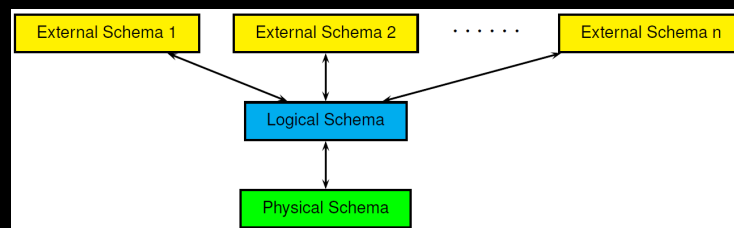


Figure 3: Levels of data abstraction.

Due to the three levels of abstraction, it is possible to insulate users/applications from changes in how the data is structured and stored, i.e., **physical** and **logical data independence** are achieved.

physical data independence: protection from changes in physical schema

logical data independence: protection from changes in logical schema

1.2.1 Transactions

A **transaction** is an abstraction for representing a logical unit of work/processing. It consists of one or more update/retrieval operations (i.e., SQL statements). Each transaction start with a **begin** command, and each transaction must end with either a **commit** or **rollback** command.

commit: all actions will be committed and reflected on the database

rollback: all actions will not be reflected in the database

DBMS adheres to the following **ACID properties** when performing transactions:

-
- **Atomicity:** either *all* the effects of a transaction are reflected in the database, or *none* are.
 - **Consistency:** the execution of a transaction in isolation *preserves the integrity constraints* of the database.
 - **Isolation:** the execution of a transaction would not interfere with other concurrent transactions to give rise to an invalid/illegal state.
 - **Durability:** the effects of a committed transaction persists in the database even in the presence of system failures.

2 RELATIONAL ALGEBRA

Lecture 2
17th January 2022

Relational algebra is a formal language for asking queries on relations. A **query** is composed of a collection of operators called **relational (algebra) operators**, each which takes 1-2 relations as input and computes an output relation. Some basic relational operators include:

- **Unary operators:** selection σ , projection π , and renaming ρ .
- **Binary operators:** cross-product \times , union \cup , intersection \cap , and difference $-$.

Relations are *closed* under relational operators, and operators can be *composed* to form **relational algebra expressions (RAEs)**. We can recursively define RAEs as below:

Let \mathcal{U} be a unary operator and \circ be a binary operator. Then,

- A relation \mathcal{R} is a RAE.
- If \mathcal{R} is a RAE, then $\mathcal{U}(\mathcal{R})$ is also an RAE.
- If \mathcal{R} and \mathcal{S} are RAEs, then $\mathcal{R} \circ \mathcal{S}$ is also an RAE.
- If \mathcal{R} is a RAE, then (\mathcal{R}) is also a RAE.

Two methods to improve the readability of relational algebra queries include:

1. operator trees, e.g.:

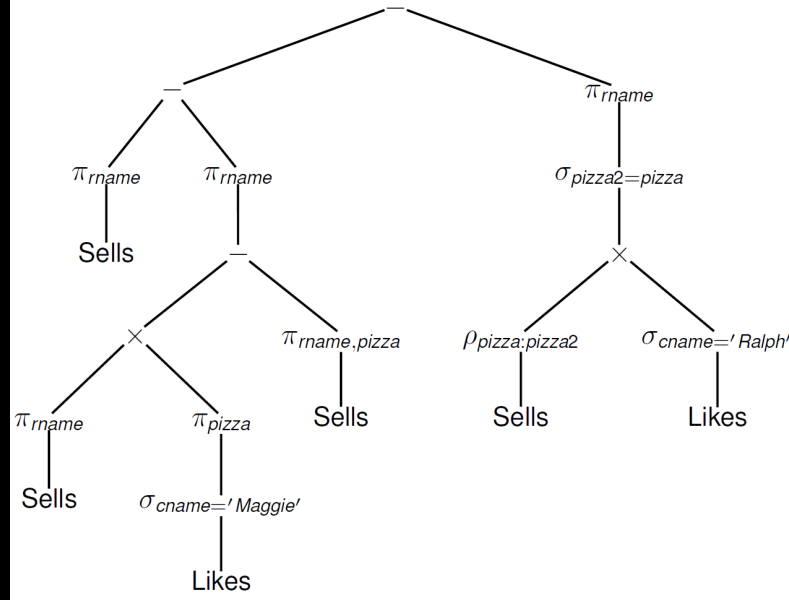


Figure 4: Example of an operator tree.

2. sequence of steps, e.g.:

$$\begin{aligned}
 R_1 &= \pi_{\text{pizza}}(\sigma_{\text{cname}='Maggie'}(\text{Likes})) \\
 R_2 &= \pi_{\text{rname}}(\text{Sells}) \times R_1 \\
 R_3 &= \pi_{\text{rname}}(R_2 - \pi_{\text{rname}, \text{pizza}}(\text{Sells})) \\
 R_4 &= \pi_{\text{rname}}(\text{Sells}) - R_3 \\
 R_5 &= \rho_{\text{pizza:pizza5}}(\sigma_{\text{cname}='Ralph'}(\text{Likes})) \\
 R_6 &= \pi_{\text{rname}}(\sigma_{\text{pizza5}=\text{pizza}}(\text{Sells} \times R_5)) \\
 \text{Answer} &= R_4 - R_6
 \end{aligned}$$

Figure 5: Example of breaking up a query into a sequence of steps.

2.1 Unary Operators

A **selection condition** is a boolean combination of terms. A **term** is one of the following forms:

- attribute **op** constant
- attribute₁ **op** attribute₂
- term₁ **and** term₂

- $\text{term}_1 \text{ or } \text{term}_2$
- **not** term_1
- (term_1)

where $\text{op} \in \{=, <>, <, \leq, >, \geq\}$. Note that the order of operator precedence is:

$<>$ is equivalent to
 $!=$ or \neq .

$$() > \text{op} > \text{not} > \text{and} > \text{or}$$

When null values are involved,

- the result of a *comparison* operation (e.g., $<>$) is **unknown**.
- the result of an *arithmetic* operation (e.g., $+$) is **null**.
- the result of *logic* operations (e.g., **and**) is described by the table below:

x	y	x AND y	x OR y	NOT x
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	TRUE	FALSE	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	TRUE	
TRUE	FALSE	FALSE	TRUE	
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	

Figure 6: Table illustrating the three-valued logic system.

2.1.1 Selection

The **selection** operator $\sigma_c(\mathcal{R})$ selects tuples from relation \mathcal{R} that satisfy the *selection condition* c . More specifically,

$$\forall t \in \mathcal{R}, t \in \sigma_c(\mathcal{R}) \text{ iff } c \text{ evaluates to } \text{true} \text{ on } t.$$

The output relation $\sigma_c(\mathcal{R})$ has the same schema as \mathcal{R} .

2.1.2 Projection

The **projection** operator $\pi_l(\mathcal{R})$ projects attributes given by a list l of attributes from relation \mathcal{R} .

The schema of the output relation is determined by l . Each attribute in l must be an attribute in \mathcal{R} , and duplicate records are removed in the output relation.

2.1.3 Renaming

The **renaming** operator $\rho_l(\mathcal{R})$ renames attributes in \mathcal{R} based on a list of attribute renamings l , of the form $a_1 : b_1, \dots, a_n : b_n$.

The schema of the output relation is the same as \mathcal{R} , except that some attributes are renamed based on l .

2.2 Binary Operators

2.2.1 Set Operators

Set operators include:

- **Union** ($\mathcal{R} \cup \mathcal{S}$): returns a relation containing all tuples that occur in \mathcal{R} , \mathcal{S} , or both.
- **Intersection** ($\mathcal{R} \cap \mathcal{S}$): returns a relation containing all tuples that occur in both \mathcal{R} and \mathcal{S} .
- **Set difference** ($\mathcal{R} - \mathcal{S}$): returns a relation containing all tuples in \mathcal{R} but not in \mathcal{S} .

These operators require input relations to be **union compatible**, i.e.:

1. they have the *same number of attributes*,
2. the corresponding attributes have the *same domains*.

The schema of the output relation is identical to the schema of \mathcal{R} (and \mathcal{S}). Note that union compatible relations do not need to have the same attribute names.

2.2.2 Cross-Product

Given relations $\mathcal{R}(A, B, C)$ and $\mathcal{S}(X, Y)$, the **cross-product** ($\mathcal{R} \times \mathcal{S}$), a.k.a. *Cartesian product*, outputs a relation with schema (A, B, C, X, Y) defined as follows:

$$\mathcal{R} \times \mathcal{S} = \{(a, b, c, x, y) | (a, b, c) \in \mathcal{R}, (x, y) \in \mathcal{S}\}$$

Note that the cross product of a relation with cardinality = 0 and a relation with a nonzero cardinality gives rise to a relation with cardinality 0.

2.3 Join Operators

Let $\text{attr}(\mathcal{R})$ denote the list of attributes in the schema of \mathcal{R} . A **dangling tuple** is a tuple in a join operand that does not participate in the join operation. More specifically, we say that $t \in \mathcal{R}$ is a dangling tuple in \mathcal{R} w.r.t. $\mathcal{R} \bowtie_c \mathcal{S}$ if $t \notin \pi_{\text{attr}(\mathcal{R})}(\mathcal{R} \bowtie_c \mathcal{S})$.

2.3.1 Inner Join

The **inner join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \bowtie_c \mathcal{S} = \sigma_c(\mathcal{R} \times \mathcal{S})$$

2.3.2 Natural Join

The **natural join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \bowtie \mathcal{S} = \pi_l(\mathcal{R} \bowtie_c \rho_{a_1:b_1, \dots, a_n:b_n}(\mathcal{S}))$$

where:

- A = the common attributes between \mathcal{R} and \mathcal{S} ,
- $c = (a_1 = b_1) \text{ and } \dots \text{ and } (a_n = b_n)$, and
- l is:
 1. the list of attributes in \mathcal{R} that are also in A , followed by
 2. the list of attributes in \mathcal{R} that are not in A , followed by
 3. the list of attributes in \mathcal{S} that are not in A .

2.3.3 Outer Join

Outer joins preserve dangling tuples in the output relation.

Let $\text{dangle}(\mathcal{R} \bowtie_c \mathcal{S})$ denote the set of dangling tuples in \mathcal{R} w.r.t. $(\mathcal{R} \bowtie_c \mathcal{S})$, and let $\text{null}(\mathcal{R})$ denote a n -component tuple of null values. Then,

- the **left outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \rightarrow_c \mathcal{S} = (\mathcal{R} \bowtie_c \mathcal{S}) \cup (\text{dangle}(\mathcal{R} \bowtie_c \mathcal{S}) \times \{\text{null}(\mathcal{S})\})$$

- the **right outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \leftarrow_c \mathcal{S} = (\mathcal{R} \bowtie_c \mathcal{S}) \cup (\{\text{null}(\mathcal{S})\} \times \text{dangle}(\mathcal{S} \bowtie_c \mathcal{R}))$$

- the **full outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \leftrightarrow_c \mathcal{S} = (\mathcal{R} \bowtie_c \mathcal{S}) \cup (\text{dangle}(\mathcal{R} \bowtie_c \mathcal{S}) \times \{\text{null}(\mathcal{S})\}) \cup (\{\text{null}(\mathcal{S})\} \times \text{dangle}(\mathcal{S} \bowtie_c \mathcal{R}))$$

The natural outer joins can be defined in a similar manner as the natural join:

- the **natural left outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \rightarrow \mathcal{S} = \pi_l(\mathcal{R} \rightarrow_c \rho_{a_1:b_1, \dots, a_n:b_n}(\mathcal{S}))$$

- the **natural right outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \leftarrow \mathcal{S} = \pi_l(\mathcal{R} \leftarrow_c \rho_{a_1:b_1, \dots, a_n:b_n}(\mathcal{S}))$$

- the **natural full outer join** of \mathcal{R} and \mathcal{S} is defined as

$$\mathcal{R} \leftrightarrow \mathcal{S} = \pi_l(\mathcal{R} \leftrightarrow_c \rho_{a_1:b_1, \dots, a_n:b_n}(\mathcal{S}))$$

3 SQL

Lecture 3
24th January 2022

SQL is a **declarative language**, i.e., it focuses on *what* to compute instead of *how* to compute. It consists of two main parts:

- **Data Definition Language (DDL)**: syntax for creating, deleting, or modifying *schemas*.
- **Data Manipulation Language (DML)**: syntax for asking queries, and inserting, deleting, or modifying *data*.

3.1 Basics

3.1.1 Data Types

Some of the built-in data types include:

- **boolean**: false, true, or null.
- **integer**: signed 4-byte integer.
- **float8**: double-precision 8-byte floating point number.
- **numeric**: arbitrary precision floating point number.
- **numeric(p, s)**: maximum total of p digits, with maximum of s digits in fractional part.
- **char(n)**: fixed-length string consisting of n characters.

- **varchar(*n*)**: variable-length string consisting up to *n* characters.
- **text**: variable-length character string.
- **date**: calendar date, i.e. (year, month, day).
- **timestamp**: date and time.

3.1.2 Tables and Comments

Comments in SQL can be written in either of the following ways:

- `-- Comments can be preceded by two hyphens`
- `/* SQL also supports C-style comments */`

To create a table in SQL, we can invoke **create table**, e.g.:

- `create table A (columns...);`

To delete a table, we can invoke **drop table**, e.g.:

- `drop table A;`
- `drop table if exists A;`
 - the **if exists** clause allows the statement to succeed even if the specified tables does not exist.
- `drop table if exists A cascade;`
 - the **cascade** option allows you to remove the table and its dependent objects.

3.1.3 Comparison Predicates

The **IS NULL** comparison predicate enables us to check if a value is equal to *null*. `x IS NOT NULL` is equivalent to `NOT (x IS NULL)`.

The **IS DISTINCT FROM** comparison predicate enables us to treat null values as ordinary values for comparison. `x IS DISTINCT FROM y`:

- is equivalent to `x <> y` if both *x* and *y* are non-null values.
- evaluates to **true** if only one of the values is null.
- evaluates to **false** if both values are null.

3.2 Constraints

SQL constraints are used to specify rules for the data in a table. They are violated if they evaluate to **false**.

By default, constraints are checked immediately *at the end* of the execution of *each SQL statement*, and any violations will cause the statement to be rolled back. However, the checking could be deferred for some constraints to the end of the execution of *a transaction*, by using the **set constraints** statement and/or specifying either:

- **deferrable initially deferred**: enforcement of constraints is deferred to the end of a transaction, or
- **deferrable initially immediate**: enforcement of constraints is deferred to the end of each statement.

```
create table Employees (
  eid          integer primary key,
  ename        varchar(100),
  managerId    integer,
  constraint employees_fkey foreign key (managerId) references Employees
                        deferrable initially immediate
);

insert into Employees values (1, 'Alice', null), (2, 'Bob', 1), (3, 'Carol', 2);

begin;
set constraints employees_fkey deferred;
delete from Employees where eid = 2;
update Employees set managerId = 1 where eid = 3;
commit;
```

Figure 7: Sample implementation of deferrable constraints.

Constraints can be specified on various levels, such as:

- column constraints, e.g.:
 - create table A (id integer **unique**);
- table constraints, e.g.:
 - create table B (id text, name text, **unique** (id, name));

The various types of constraints include:

- not-null constraints, e.g.:
 - create table A (id integer **not null**);
 - violated if $\exists x \in A$ where $x.id$ IS NOT NULL evaluates to false.

- unique constraints, e.g.:
 - create table A (id integer **unique**);
 - violated if $\exists x, y \in A$ where $x.id \neq y.id$ evaluates to false.
- primary key constraints, e.g.:
 - create table A (id integer **primary key**);
 - we can specify:


```
create table A ( id integer unique not null );
```

for other candidate keys, since only one column can be specified as the primary key in SQL.
- foreign key constraints, e.g.:
 - create table A (id integer **references** B);
 - * the column of B need not be specified if id is referring to the primary key of B.
 - * otherwise, we can specify the column(s) which we are referring to, e.g.:


```
... foreign key (id) references B (cid) ...
```
 - note that in SQL, we can refer to a (set of) column(s) as long as the (set of) column(s) has a uniqueness constraint applied to them; it doesn't need to be the primary key of a table.
 - we can use **MATCH FULL** to prevent reference to completely or partially null entries.
 - we can also specify **ON UPDATE/DELETE <ACTION>** to deal with any violation that might occur. Some possible **ACTIONS** include:
 - * **NO ACTION**: rejects delete/update if a constraint is violated (default option).
 - * **RESTRICT**: similar to NO ACTION, except that constraint checking cannot be deferred.
 - * **CASCADE**: propagates delete/update to referencing tuples.
 - * **SET DEFAULT**: updates foreign keys of referencing tuples to some default value.
 - * **SET NULL**: updates foreign keys of referencing tuples to null.
- general constraints, such as:
 - check constraints, e.g.:
 - * create table A (id integer **check** (id < 5));

We can also name our constraints so that error messages will show the name of the constraint that is violated, and it will also be easier to remove constraints from our database later on.

3.2.1 Assertions

SQL's **create assertion** statement supports the specification of general constraints. However, this statement is not implemented in many DBMSes, and **triggers** are often used to enforce general constraints instead.

3.3 Database Modifications

After creating a table, we can insert entries into the table by invoking `insert into`, e.g.:

- `insert into B values (123, 'Hello');` or
- `insert into B values (123);`
 - if we do not specify the value for a column, the default value for the column (usually null) will be assigned to the new entry.
 - we can modify the default value of a column by using the **default** statement, e.g.:
* `create table B (id integer, name text default 'CS');`

We can also remove entries from the table by invoking `delete from`, e.g.:

- `delete from A;`
 - removes all entries from A.
- `delete from A where id = 3;`
 - removes all entries with `id = 3` from A.

Additionally, we can update the entries in the table by invoking `update/set`:

- `update A set id = id * 2;` or
- `update B set name = 'Hello' where id = 1;`
 - updates the name of the entry with `id = 1` to 'Hello'.

Furthermore, we can add/remove/modify columns or add/remove constraints by invoking `alter table`, e.g.:

- `alter table B alter column name drop default;`
- `alter table B drop column name;`
- `alter table B add column day date;`
- `alter table B add constraint fk_id foreign key (id) references A;1`

¹refer to this link for formatting.

3.4 Queries

Lecture 4
7th February 2022

SQL is also a **data manipulation language (DML)** (i.e., it contains commands permitting users to manipulate data in a database). The basic form of an SQL query consists of three clauses:

- `select [distinct] a_1, \dots, a_m [as col_name]`
 - `distinct` indicates whether duplicate records would be kept in the output relation.
 - a_1, \dots, a_m specifies columns to be included in output table.
 - * * indicates that all columns would be selected.
 - * the operator `||` can be used to concatenate strings.
 - * `round()` can be used to round off a value to the nearest integer.
 - `as col_name` renames the selected column to `col_name` in the new table.
 - set operations (e.g., `union`, `intersect`, `except`) can be used on select clauses.
 - * appending the term “`all`” to a set operation preserves duplicate records; otherwise, duplicate records are removed by default.
- `from r_1, \dots, r_n`
 - r_1, \dots, r_n specifies the list of relations.
 - * a comma “,” is equivalent to a *cross join* between two relations.
 - * we can rename the relations using *as*, e.g., “from Customers as C”, or simply as “from Customers C”.
 - * we can also perform a *join* (or equivalently an *inner join*) on two relations, based on a column.
 - * we can also perform a *natural join*, a *left/right/full (outer) join*, or a *natural left/right/full (outer) join* on two relations.
- `[where c]`
 - c specifies the conditions on the relations.

These three clauses together are equivalent to the relational algebra expression

$$\pi_{a_1, \dots, a_m} \left(\sigma_c (r_1 \times r_2 \times \dots \times r_n) \right)$$

3.4.1 Subqueries

The main subquery expressions in SQL includes:

- **EXISTS** subqueries: returns true if the output of the subquery is non-empty, e.g.:

- `select ... where [not] exists(select 1 from ...);`
- we usually use “select 1” in exists subqueries because we only care about whether the output of the inner query is empty or not (and not the actual output).
- **IN** subqueries: returns false if the output of the subquery is empty, else returns the result of the expression $(v = v_1) \vee (v = v_2) \vee \dots \vee (v = v_n)$.
 - Aside from the format as seen in exists subqueries, in subqueries may also adopt the format:


```
select ... where X in(value1, ..., valuen);
```
- **ANY/SOME** subqueries: returns false if the output of the subquery is empty, else returns the results of the boolean expression $(v \text{ op } v_1) \vee (v \text{ op } v_2) \vee \dots \vee (v \text{ op } v_n)$.
 - in other words, any subqueries are a general form of in subqueries, where any operators on top of “=” may be used.
- **ALL** subqueries: returns true if the output of the subquery is empty, else return $(v \text{ op } v_1) \wedge (v \text{ op } v_2) \wedge \dots \wedge (v \text{ op } v_n)$.

It is possible to use subqueries that return more than one column, by using a row constructor, e.g.:

- ```
select a, b, c
from Table T
where row(a, b) = all(...);
```

Non-scalar subquery expressions can be used in different parts of SQL queries, including:

- the `where` clause,
- the `from` clause, and
  - subqueries in the from clause must be *enclosed in parentheses* and *assigned a table alias*, and optionally with *column aliases*.
  - we can also use *values* here to generate a temporary table used for the query operation.
- the `having` clause.

### 3.4.2 Scalar Subqueries

A **scalar subquery** is a subquery that returns at most one tuple with one column. If the subquery’s output is empty, its return value is null.

### 3.4.3 Additional Query Clauses

On top of `select`, `from`, and `where`, SQL queries may also consist of the following optional clauses:

- `order by`  $X$  (**asc**),  $Y$  (**desc**)
- `limit`  $n$ : returns the top  $n$  tuples based on the current ordering (if any).
- `offset`  $n$ : excludes the top  $n$  tuples from the table based on the current ordering (if any).

### 3.4.4 Aggregate Functions

Lecture 5  
15<sup>th</sup> February 2022

An **aggregate function** computes a single value from a set of tuples. Some of the basic aggregate functions (i.e., **min**, **max**, **avg**, **sum**, **count**) are described in the diagram below:

| Query                                        | Meaning                                       |
|----------------------------------------------|-----------------------------------------------|
| <code>select min(A) from R</code>            | Minimum non-null value in A                   |
| <code>select max(A) from R</code>            | Maximum non-null value in A                   |
| <code>select avg(A) from R</code>            | Average of non-null values in A               |
| <code>select sum(A) from R</code>            | Sum of non-null values in A                   |
| <code>select count(A) from R</code>          | Count number of non-null values in A          |
| <code>select count(*) from R</code>          | Count number of rows in R                     |
| <code>select avg(distinct A) from R</code>   | Average of distinct non-null values in A      |
| <code>select sum(distinct A) from R</code>   | Sum of distinct non-null values in A          |
| <code>select count(distinct A) from R</code> | Count number of distinct non-null values in A |

Figure 8: Table illustrating some of the basic aggregate functions.

Aggregate functions can be used in different parts of SQL queries, including the `SELECT`, `HAVING`, and `ORDER BY` clauses. However, they cannot be used in `WHERE` clauses.

### 3.4.5 GROUP BY clause

The `GROUP BY` clause allows us to partition tuples into groups based on an attribute. More specifically, in a query with `group by`  $a_1, a_2, \dots, a_n$ , two tuples  $t, t'$  are said to belong to the same group if the following expression evaluates to true:

$$(t.a_1 \text{ is not distinct from } t'.a_1) \wedge \dots \wedge (t.a_n \text{ is not distinct from } t'.a_n)$$

However, for a query involving a `GROUP BY` clause to be valid, one of the

following conditions must hold for each column  $\mathcal{A}$  in relation  $\mathcal{R}$  that appears in the **SELECT** clause:

1.  $\mathcal{A}$  appears in the **GROUP BY** clause.
2.  $\mathcal{A}$  appears in an aggregated expression in the **SELECT** clause.
3. the primary (or candidate, for non-psql systems) key of  $\mathcal{R}$  appears in the **GROUP BY** clause.

Additionally, if an aggregate function appears in the **SELECT** clause and there is not **GROUP BY** clause, then the **SELECT** clause *must not* contain any column that is *not in an aggregated expression*.

### 3.4.6 HAVING clause

The **HAVING** clause allows us to specify a boolean condition which each of the groups in the output relation must satisfy. For a query involving a **HAVING** clause to be valid, one of the following conditions must hold for each column  $\mathcal{A}$  in relation  $\mathcal{R}$  that appears in the **HAVING** clause:

1.  $\mathcal{A}$  appears in the **GROUP BY** clause.
2.  $\mathcal{A}$  appears in an aggregated expression in the **HAVING** clause. (this is the only condition that differs from the conditions in the previous section)
3. the primary (or candidate, for non-psql systems) key of  $\mathcal{R}$  appears in the **GROUP BY** clause.

### 3.4.7 Conceptual Evaluation of Queries

For a query with the given structure, it will be evaluated in the following order:

|                 |                             |
|-----------------|-----------------------------|
| <b>select</b>   | <b>distinct</b> select-list |
| <b>from</b>     | from-list                   |
| <b>where</b>    | where-condition             |
| <b>group by</b> | groupby-list                |
| <b>having</b>   | having-condition            |
| <b>order by</b> | orderby-list                |
| <b>offset</b>   | offset-specification        |
| <b>limit</b>    | limit-specification         |

1. Cross-product of the tables in the **from-list**.
2. Select tuples that evaluate to true for the **where-condition**.
3. Partition selected tuples using the **groupby-list**.

4. Select groups that evaluate to true for the **having-condition**.
5. Generate an output tuple for each selected group based on the expressions in the **select-list**.
6. Remove duplicate output tuples (if **distinct** is specified).
7. Sort output tuples based on the **orderby-list**.
8. Remove output tuples based on the **offset-specification** and **limit-specification**.

### 3.5 Advanced Features

#### 3.5.1 Common Table Expressions (CTEs)

We can use CTEs with the following format:

```
with
 R1 as (Q1),
 R2 as (Q2),
 ...,
 Rn as (Qn)
select/insert/update/delete statement S;
```

Figure 9: SQL format for CTEs. Note that the last CTE (i.e.,  $R_n$ ) is not succeeded by a comma.

to specify temporary tables, which could be used in queries. CTEs are especially useful for writing recursive queries.

#### 3.5.2 Views

A **view** defines a virtual relation that can be used for querying (refer to Figure 3). Views help to ensure **logical (physical) data independence**, i.e., insulating users/applications from changes to the underlying logical (physical) schema.

We can use the following format to create a view:

```
create view CourseInfo as
select cname, pname, lectureTime,
 numUGrad+numPGrad+numExchange+numAudit as numEnrolled
from Courses natural join Profs natural join Enrollment;

create view CourseInfo (cname, pname, lectureTime, numEnrolled) as
select cname, pname, lectureTime,
 numUGrad + numPGrad + numExchange + numAudit
from Courses natural join Profs natural join Enrollment;
```

Figure 10: Two commands which creates the same view.

### 3.5.3 Conditional Expressions

SQL also allows for the use of conditional expressions, such as:

- **CASE**: similar to the case statement in other programming languages. Formats:

```
– select case
 when <condition> then <result>
 :
 when <condition> then <result>
 else <result>
 end as <attributeName>
– select case <expression>
 when <value> then <result>
 :
 when <value> then <result>
 else <result>
 end as <attributeName>
```

- **COALESCE**: returns the first non-null value in its arguments. Format:

```
– select coalesce(arg1, ..., argn) as <attributeName>
```

- **NULLIF**: returns null if *value*<sub>1</sub> is equal to *value*<sub>2</sub>, else returns *value*<sub>1</sub>. Format:

```
– select nullif(value1, value2) as <attributeName>
```

### 3.5.4 Pattern Matching

The **LIKE** operator enables us to match strings with a particular pattern.

```
select \mathcal{A} from \mathcal{R} where \mathcal{A} like '%_'
```

- **\_** matches any single character.
- **%** matches any sequence of 0 to more characters.

## 4 CONCEPTUAL DATABASE DESIGN

Lecture 6  
31<sup>st</sup> January 2022

The purpose of **conceptual data models** is to capture data requirements using a conceptual schema. Some models include:

- the **Entity-Relationship (ER)** model, and
- the **Unified Modelling Language (UML)**.

In this module, we will be focusing on the ER model.

### 4.1 Entity-Relationship (ER) Model

In the ER model, data is described using **entities** and **relationships**. Information about entities and relationships are described using **attributes**. ER schemas are presented as *ER diagrams*.

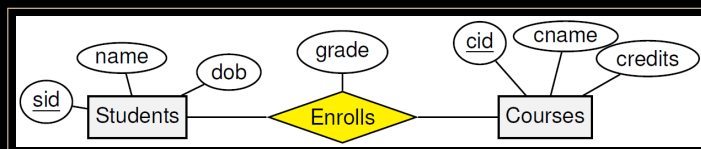


Figure 11: Example of an ER diagram. Entity sets are represented by rectangles, attributes are represented by ovals, and relationship sets are represented by diamonds. The attributes that form a primary key of an entity set are underlined.

**entity:** real-world object distinguishable from other objects

**entity set:** collection of similar entities

**relationship:** association among two or more entities

**relationship set:** collection of similar relationships

**attribute:** specific information describing an entity or a relationship

Each entity set participating in a relationship set plays a certain **role**, which is typically named the same as the entity set name and is not shown explicitly. Roles are only shown explicitly when one entity set appears two or more times in a relationship set, e.g.:

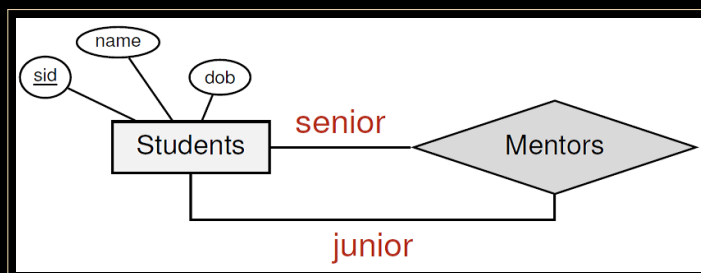


Figure 12: Example where relationship roles are explicitly shown.

An  $n$ -ary relationship set involves  $n$  entity roles, where  $n$  is also called the **degree** of the relationship set. Both examples provided above are 2-ary (a.k.a. binary) relationship sets.

Consider an  $n$ -ary relationship set  $R$  involving entity sets  $E_1, \dots, E_n$  with

relationship attributes  $\{A_1, \dots, A_n\}$ , and let  $key(E_i)$  denote the set of attributes that define the *primary key of entity set*  $E_i$ . Then,  $key(R)$  is specified by some subset  $A' \subseteq \{A_1, \dots, A_k\}$  and some subset  $E' \subseteq \{E_1, \dots, E_n\}$  such that

$$key(R) = A' \cup \left( \bigcup_{E_i \in E'} key(E_i) \right)$$

is a minimal subset of attributes whose values uniquely identify a relationship instance of  $R$ .

#### 4.1.1 Relationship Constraints

Let  $R$  be a relationship set that involves entity set  $E$ . Then,

- the **key constraint** on  $E$  w.r.t.  $R$  implies that each instance of  $E$  can participate in *at most one* instance of  $R$ . In ER diagrams, this is indicated using a directed line, e.g.:

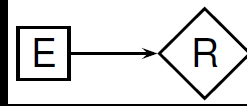


Figure 13: Example of a key constraint.

- the **participation constraint** on  $E$  w.r.t.  $R$  determines if the participation of an entity set in a relationship set is mandatory.
  - the **total participation constraint** implies that each instance of  $E$  must participate in *at least one* instance of  $R$ . In ER diagrams, this is indicated using a bold line, e.g.:

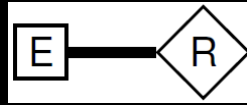


Figure 14: Example of a total participation constraint.

- the **partial participation constraint** implies that each instance of  $E$  can participate in *0 or more* instances of  $R$ . In ER diagrams, this is indicated using a normal (i.e., non-bold) line.

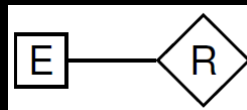


Figure 15: Example of a partial participation constraint.



### 4.1.2 Weak Entity Sets

A **weak entity set** does not have its own key. Instead, it only contains a **partial key**, i.e., a set of attributes of the entity set that uniquely identifies a weak entity for a given owner entity. A weak entity can only be uniquely identified by considering the primary key of another entity, called the **owner entity**.

- there must be a *many-to-one relationship* (i.e., **identifying relationship**) from the weak entity set to an owner entity set
- the weak entity set must also have *total participation* in the identifying relationship.
- in ER diagrams, this is indicated using a bold diamond and entity, along with a bold directed line, e.g.:

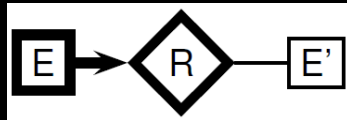


Figure 16: Example of a weak entity set, E, with identifying owner E' and identifying relationship set R.

A weak entity's existence is dependent on the existence of its owner entity; if an owner entity is deleted, all the weak entities that are dependent on the owner entity will also be deleted.

### 4.1.3 Aggregation

**Aggregations** can be used to model relationships between an entity set and a relationship set. In ER diagrams, this is indicated using a bold square, e.g.:

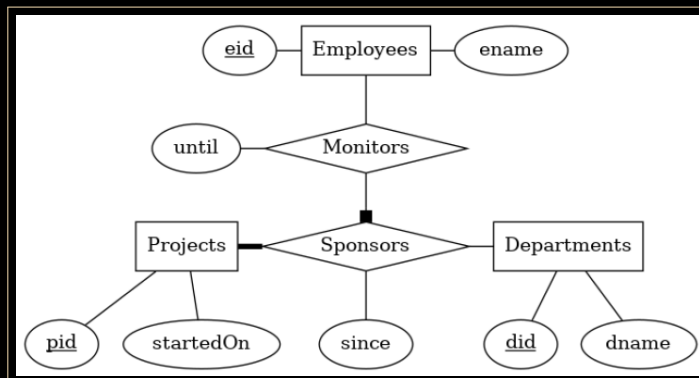


Figure 17: The ■ connected to Sponsors denotes that the Sponsors relationship set is participating in the Monitors relationship as an aggregation.

## 4.1.4 ISA Hierarchies

ISA (i.e., is a) **hierarchies** help to classify entity sets into subclasses. Every entity in a subclass entity set is an entity in its superclass entity set.

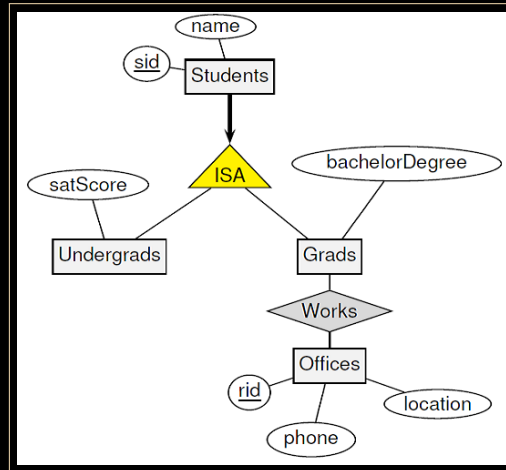


Figure 18: Example of an ISA hierarchy.

There are some constraints associated with ISA hierarchies, including:

- **overlap constraint:** an ISA hierarchy satisfies this constraint if an entity in a superclass could belong to multiple subclasses.
- **covering constraint:** an ISA hierarchy satisfies this constraint if every entity in a superclass *has to belong* to some subclass.

Figure 18 illustrates how different combinations of the above constraints could be illustrated in ER diagrams.

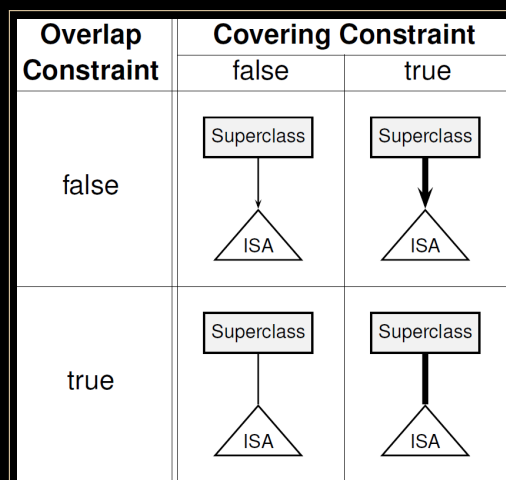


Figure 19: ISA hierarchy constraints.

## 5 PROCEDURAL SQL

Lecture 7  
28<sup>th</sup> February 2022

In previous lectures, we have learnt that SQL is both a data definition language (DDL) and a data manipulation language (DML). However, we would like to utilize SQL as part of a procedural language.

| Declarative                                                                                     | Procedural                                                |
|-------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| Specifies the “what”.                                                                           | Specifies the “how”.                                      |
| Requires less lines for solving <i>generic queries</i> (e.g., find student with highest grade). | Requires less lines for solving <i>specific queries</i> . |

Figure 20: Table illustrating the differences between declarative and procedural languages.

There are two ways to achieve this:

1. Using a host language + SQL.
2. Using PL/pgSQL.

### 5.1 Host Language + SQL

We will use the C language as an example. There are two types of mixing, namely:

- Mixing at the **statement-level interface** (i.e., C + SQL).
- Mixing at the **call-level interface** (i.e., C only).

At the statement-level interface, the basic idea consists of:

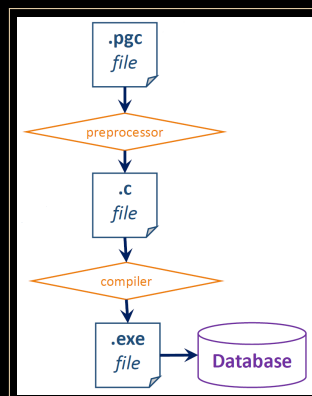


Figure 21: Procedure at the statement-level interface.

1. **Write** a program that mixes the host language with SQL (e.g., in .pgc format).
2. **Preprocess** the program using a preprocessor.
3. **Compile** the program into an executable code.

In this case, we could utilize either static SQL (i.e., with fixed queries) or dynamic SQL (i.e., which generates queries at runtime).

At the call-level interface, the basic idea consists of:

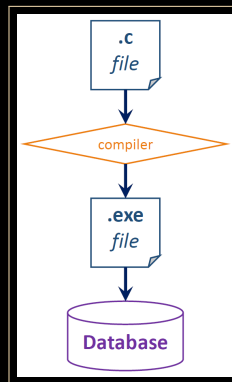


Figure 22: Procedure at the call-level interface.

1. **Write** in host language only.
2. **Compile** the program into an executable code.

In this case, we will need to load a library that provides APIs to access the database (e.g., ODBC, which contains methods such as connection, work, and disconnect).

## 5.2 PL/pgSQL

**PL/pgSQL** is a SQL-based procedural language for PostgreSQL, which standardizes the syntax and semantics of the SQL procedural language.

In PL/pgSQL, we can define a **function** as follows:

- Functions are useful because they allow for:
  - code reuse.
  - ease of maintenance.
  - performance.
- To call a function, we could use `SELECT <function name>(<param>)` or `SELECT * from <function name>(<param>)`

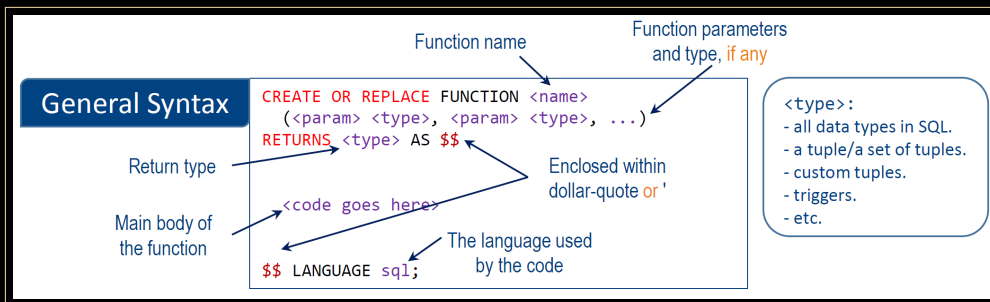


Figure 23: Sample function definition in PL/pgSQL.

- To return a tuple, we can change the return `<type>` to the name of the table.
  - To return more than one tuple, we can utilize the notation `SETOF`.
- We could also specify the input and output names + datatypes in the parameters, using `IN` and `OUT`.
  - For custom tuples, we could simplify the parameters by simply omitting the parameter for the function, and replacing the return `<type>` to `TABLE(...)`.
  - A function can also return nothing by replacing the return `<type>` with `VOID`.

Aside from functions, PL/pgSQL also supports **procedures**, which can be defined as follows:

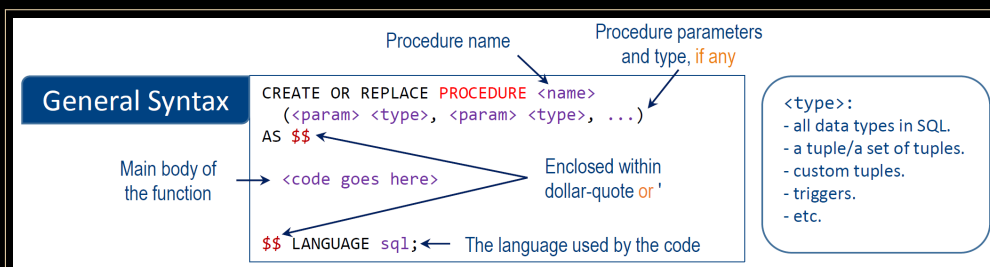
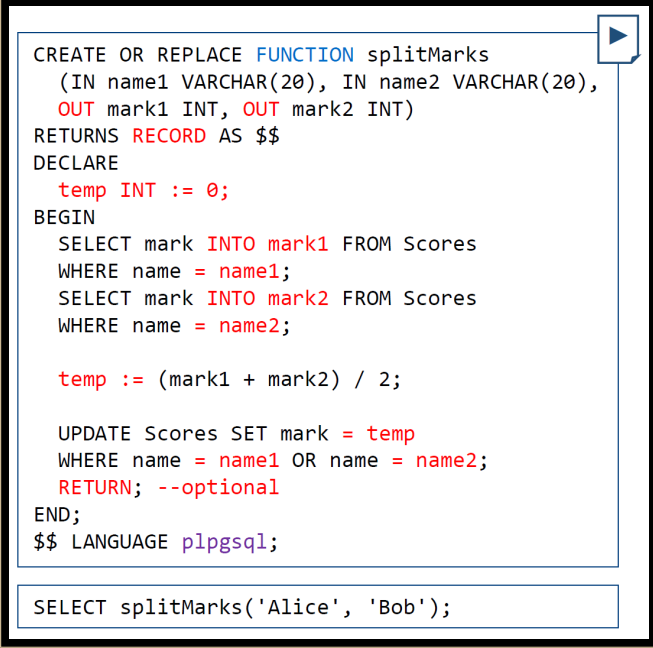


Figure 24: Sample procedure definition in PL/pgSQL.

- Note that SQL procedures *do not have a return value*.
- To call a procedure, we could use `CALL <procedure name>(<param>)`.

### 5.2.1 Variables

In PL/pgSQL, there are **variables** and **control structure**. We can declare variables using the `DECLARE` notation along with `BEGIN` denoting the end of the declaration portion, and assign variables using either `:=` or `INTO`.



```

CREATE OR REPLACE FUNCTION splitMarks
 (IN name1 VARCHAR(20), IN name2 VARCHAR(20),
 OUT mark1 INT, OUT mark2 INT)
RETURNS RECORD AS $$
DECLARE
 temp INT := 0;
BEGIN
 SELECT mark INTO mark1 FROM Scores
 WHERE name = name1;
 SELECT mark INTO mark2 FROM Scores
 WHERE name = name2;

 temp := (mark1 + mark2) / 2;

 UPDATE Scores SET mark = temp
 WHERE name = name1 OR name = name2;
 RETURN; --optional
END;
$$ LANGUAGE plpgsql;

SELECT splitMarks('Alice', 'Bob');

```

Figure 25: An example of variable usage in PL/pgSQL.

To return a set of custom tuples, we could use the notation `RETURN NEXT/QUERY`. Note that the usage of return statements does not exit the function.

### 5.2.2 Control Structure

In PL/pgSQL, we can also manipulate the control flow of the code using conditional statements and loops. The list of control structure manipulation supported by PL/pgSQL include:

- Conditional statements:
  - `IF ... ELIF ... THEN ... ELSE ... END IF;`
- Loops:
  - `LOOP`
  - `...`
  - `END LOOP;`
  - `WHILE ... LOOP`
  - `...`
  - `END LOOP;`
  - `FOR (EACH) ... IN ... LOOP`
  - `...`

END LOOP;

- Breaks:

- EXIT WHEN ...

### 5.2.3 Cursors

A **cursor** enables us to *access each individual row* returned by a select statement. The workflow of a cursor is illustrated in figure 26 below.

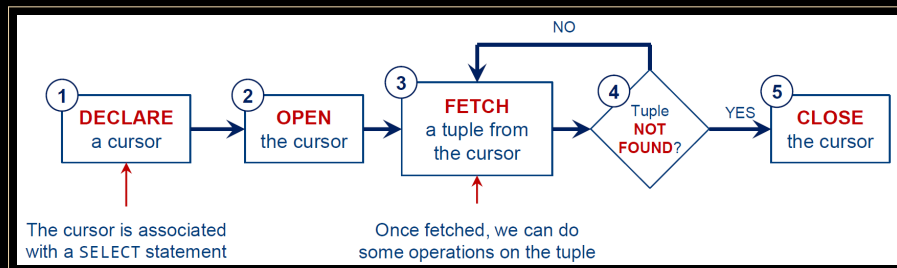


Figure 26: Workflow of a cursor. Note that other statements could be used at step 3, including MOVE, UPDATE, and/or DELETE.

Figure 27 below illustrates a sample use of cursors.

```

CREATE OR REPLACE FUNCTION consCryptosDown
(IN n INT)
RETURNS TABLE(rank INT, sym CHAR(4)) AS $$
DECLARE
 curs CURSOR FOR (SELECT * FROM cryptosRank
 WHERE changes < -5);
 r1 RECORD;
 r2 RECORD;
BEGIN
 OPEN curs;

 LOOP
 <code snippet goes here>
 END LOOP;

 CLOSE curs;
END;
$$ LANGUAGE plpgsql;

```

```

FETCH curs INTO r1;
EXIT WHEN NOT FOUND;

FETCH RELATIVE (n-1)
FROM curs INTO r2;
EXIT WHEN NOT FOUND;

IF r2.rank - r1.rank = n-1 THEN
 MOVE RELATIVE -(n) FROM curs;

 FOR c IN 1..n LOOP
 FETCH curs INTO r1;
 rank := r1.rank;
 sym := r1.symbol;
 RETURN NEXT;
 END LOOP;

 CLOSE curs;
RETURN;
END IF;

MOVE RELATIVE -(n-1) FROM curs;

```

| Rank | Symbol | Changes |
|------|--------|---------|
| 1    | BTC    | -6%     |
| 3    | DOGE   | -6%     |
| 4    | ZIL    | -7%     |
| 5    | XMR    | -8%     |
| 6    | SHIB   | -8%     |
| 8    | LTC    | -7%     |
| 9    | XRP    | -7%     |
| 10   | BNB    | -6%     |

Figure 27: Sample use of cursor to find the first 3 consecutive coins that are down by > 5%.

The syntax for cursors is listed below:

- FETCH [PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n] FROM <cursor> INTO <var>;
- MOVE [PRIOR | FIRST | LAST | ABSOLUTE n | RELATIVE n] FROM <cursor>;
- [UPDATE | DELETE] <table> ... WHERE CURRENT OF <cursor>;

### 5.3 SQL Injection

One reason for using PL/pgSQL over host language + SQL is to protect the database against attacks on dynamic SQL (i.e., SQL injections). For example, malicious inputs may include inverted commas (') to escape a string, and introduce malicious code into the database.

Some ways to protect the database includes:

- Using a function or procedure.

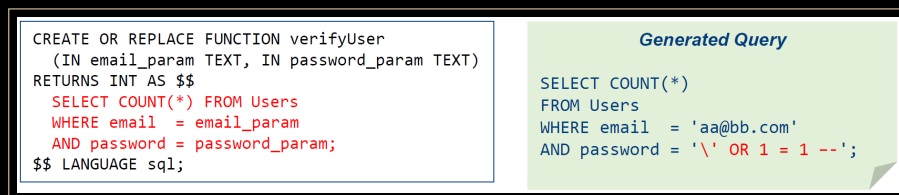


Figure 28: Example of using SQL functions to protect against injections.

- By doing so, all inputs will be treated as strings at runtime, since functions and procedures are compiled and stored in the database.
- Using PREPARE statements.

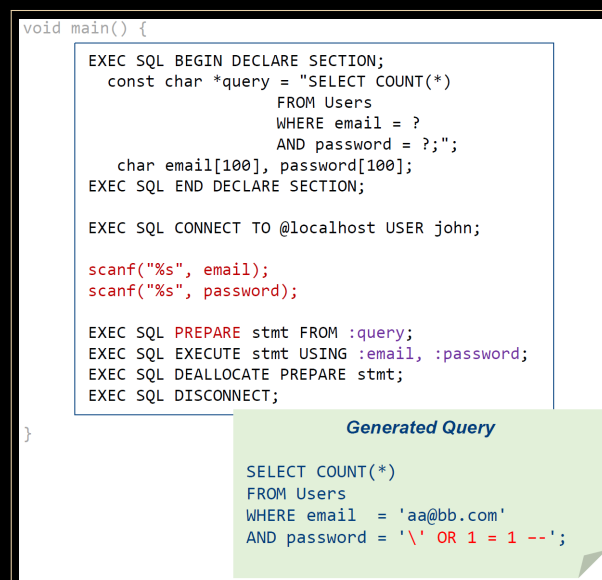


Figure 29: Example of using PREPARE statements to protect against injections.

- By doing so, the SQL query is compiled before it is prepared, so all inputs are treated as strings at runtime.



---

## 6 TRIGGERS

Lecture 8  
7<sup>th</sup> March 2022

Triggers can be used to tell the database to:

- watch out for insertions on a table, and
- call a trigger function after each insertion of a tuple.

The format of a trigger is as follows:

- CREATE TRIGGER <trigger\_name>  
  
[AFTER | BEFORE | INSTEAD OF] [INSERT | DELETE | UPDATE] ON  
<table\_name>  
  
FOR EACH [ROW | STATEMENT]  
  
EXECUTE FUNCTION <function\_name>();

On the other hand, a trigger function has the following format:

- CREATE OR REPLACE FUNCTION <function\_name>()  
  
RETURNS TRIGGER AS \$\$  
  
BEGIN  
  
...  
  
RETURN X;  
  
END;  
  
\$\$ LANGUAGE plpgsql;
- Inside a trigger function, we can utilize the following:
  - NEW: refers to the new row inserted into the table.
  - OLD: refers to the old tuple being updated/deleted.
  - CURRENT\_DATE: returns the current date.
  - TG\_OP: the operation that activates the trigger (e.g., INSERT, UPDATE, or DELETE).
  - TG\_TABLE\_NAME: the name of the table that caused the trigger invocation.

### 6.1 Trigger Timing

In a trigger, we could specify when should the trigger function be executed. The options include:

- **AFTER**: the trigger function is executed after an insertion / deletion / update.
  - For these triggers, the return value does not matter (because the trigger function is invoked after the main operation is done).
- **BEFORE**: the trigger function is executed before an insertion / deletion / update.
  - For these triggers, if the return value is NULL, no action (i.e., insertion / deletion / update) will be performed.
  - If the return value is a non-null tuple  $t$ ,  $t$  will be inserted / updated accordingly. However, in the case of deletion, the non-null return value will be ignored, and deletion proceeds as normal.
- **INSTEAD OF**: the trigger function is executed instead of the insertion / deletion / update.
  - These triggers can be **defined on views only**.
  - The typical usage involves re-performing an action on a table when the action is attempted on the view.
  - For these triggers, if the return value is NULL, the database is signalled to ignore the rest of the operations on the current row.
  - If the return value is a non-null tuple, the database will proceed as normal.

Note that for **INSERT**, OLD is initially set to NULL, so RETURN OLD is the same as RETURN NULL

### 6.2 Trigger Levels

We can also specify when should a trigger function be executed. The options include:

- **ROW**: executes the trigger function for every tuple encountered.
  - **INSTEAD OF** is only allowed on the row-level, whereas **BEFORE** and **AFTER** are allowed on both levels.
- **STATEMENT**: executes the trigger function only once.
  - Statement-level triggers ignore the values returned by trigger functions (i.e., RETURN NULL would not make the database omit subsequent operations).
  - To omit subsequent operations, we need to employ RAISE EXCEPTION <text> instead of RAISE NOTICE <text>.

### 6.3 Trigger Conditions

We can specify conditions in the trigger definition as follows:

```
• CREATE TRIGGER <trigger_name>

 [AFTER | BEFORE | INSTEAD OF] [INSERT | DELETE | UPDATE] ON
 <table_name>

 FOR EACH [ROW | STATEMENT]

 WHEN <condition>

 EXECUTE FUNCTION <function_name>;
```

However, trigger conditions are subject to the following requirements:

- No SELECT in WHEN().
- No OLD in WHEN() for INSERT.
- No NEW in WHEN() for DELETE.
- No WHEN() for INSTEAD OF.

### 6.4 Deferred Triggers

There are scenarios where we may need to defer the checking of triggers (e.g., in bank transfers, where triggers are used to ensure that the total balance of all accounts should be  $\geq n$ ). We can define a deferred trigger as follows:

```
• CREATE CONSTRAINT TRIGGER <trigger_name>

 AFTER [INSERT | DELETE | UPDATE] ON <table_name>

 DEFERRABLE [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

 FOR EACH ROW

 EXECUTE FUNCTION <function_name>;
```

Note that:

- INITIALLY DEFERRED indicates that the trigger is deferred by default.
- INITIALLY IMMEDIATE indicates that the trigger is not deferred by default.
- Deferred triggers only work with AFTER and FOR EACH ROW.

Subsequently, we can enclose the updates as a transaction, e.g.:

```
• BEGIN TRANSACTION;

 SET CONSTRAINTS <trigger_name> DEFERRED;

 UPDATE ...;

 UPDATE ...;

 COMMIT;
```

Here, the trigger will only be activated at the COMMIT step.

### 6.5 Multiple Triggers

There can be multiple triggers defined for the same event on the same table. In such situations, the order of trigger activation is as follows:

1. **BEFORE statement-level** triggers.
2. **BEFORE row-level** triggers.
3. **AFTER row-level** triggers.
4. **AFTER statement-level** triggers.

Within each category, triggers are activated in alphabetical order. Also note that if a **BEFORE row-level** trigger returns NULL, all subsequent triggers on the same row will be omitted.

## 7 NORMAL FORMS (I)

Lecture 9  
14<sup>th</sup> March 2022

How do we decide which relational schema is better? One way to approach this question is to utilize a **normal form**, which is a definition of minimum requirements to:

- reduce **data redundancy**, and
- improve **data integrity**, e.g., avoiding:
  - **update anomalies**: updating of an entry results in anomalies in the database.
  - **deletion anomalies**: deletion of an entry results in anomalies.
  - **insertion anomalies**: insertion of an entry results in anomalies.

The **normalization** (i.e., decomposition) of a table would help us get rid of these anomalies.

### 7.1 Functional Dependencies (FD)

Let  $A_1, \dots, A_m, B_1, \dots, B_n$  be some attributes. We say that  $A_1 \dots A_m \rightarrow B_1 \dots B_n$  (i.e., “ $A_1 \dots A_m$  decides/determines  $B_1 \dots B_n$ ”) if whenever two objects have the same values on  $A_1, \dots, A_m$ , they always have the same values on  $B_1, \dots, B_n$ . Note that an FD may hold on one table but not on another.

FDs can be generated from:

- common sense
- the application’s requirements

#### 7.1.1 FD Reasoning

Given a set of FDs, we may want to figure out what other FDs can they imply. To do this, we can turn to several axioms and rules:

- **Armstrong’s axioms:**
  - **Axiom of reflexivity:**  $AB \rightarrow A$ .
  - **Axiom of augmentation:** if  $A \rightarrow B$ , then  $AC \rightarrow BC$ .
  - **Axiom of transitivity:** if  $A \rightarrow B$  and  $B \rightarrow C$ , then  $A \rightarrow C$ .
- **Rule of decomposition:** if  $A \rightarrow BC$ , then  $A \rightarrow B$  and  $A \rightarrow C$ .
- **Rule of union:** if  $A \rightarrow B$  and  $A \rightarrow C$ , then  $A \rightarrow BC$ .

#### 7.1.2 Closure

To speed up FD reasoning, a more convenient approach is the use of closures.

Let  $S = \{A_1, \dots, A_n\}$  be a set of attributes. The **closure** of  $S$  (i.e.,  $\{A_1, \dots, A_n\}^+$ ) is the set of attributes that can be decided by  $A_1, \dots, A_n$ , either directly or indirectly. More specifically, the closure  $\{A_1, \dots, A_n\}^+$  can be computed as follows:

1. Initialize the closure to  $\{A_1, \dots, A_n\}$ .
2. If there is a FD (e.g.,  $A_i, \dots, A_{i+j} \rightarrow B$ ) such that  $A_i, \dots, A_{i+j} \in \{A_1, \dots, A_n\}^+$ , we put  $B$  into the closure.
3. Step 2 is repeated until no new attributes can be added to the closure.

With closures, to prove that  $X \rightarrow Y$  holds, it suffices to show that  $Y \in \{X\}^+$ . On the other hand, to show the converse, it suffices to show that  $Y \notin \{X\}^+$ .

## 7.2 Keys, Superkeys, and Prime Attributes

The keys and superkeys defined in this section are different from the keys and superkeys for entity sets, since the keys of a table are decided solely by the functional dependencies of the table. Specifically,

- a **superkey** is a set of attributes in a table that decides all other attributes, whereas
- a **key** is a minimal superkey.
  - Note that similar to entity sets, a table may also have multiple keys.
- if an attribute appears in a key, then it is known as a **prime attribute**; otherwise, it is a **non-prime attribute**.

There exists an algorithm for finding keys, which consists of the following steps:

1. Given a table  $T(A_1, \dots, A_n)$  and a set of FDs on  $T$ , consider every subset of attributes in  $T$  (i.e., all possible combinations of  $A_i \in T$ ).
2. Derive the closure of each subset (i.e.,  $\{A_1\}^+, \dots, \{A_n\}^+, \{A_1 A_2\}^+, \dots$ ).
3. Identify all superkeys based on the closures.
4. Identify all keys from the superkeys.

Some tricks which we could employ to speed up this algorithm include:

- Check for attribute sets in order of size (i.e., from smallest to largest sets).
- If an attribute does not appear in the R.H.S. of any FD, then it must be in every key.

## 8 NORMAL FORMS (II)

Lecture 10  
28<sup>th</sup> March 2022

**Normal forms** state the conditions which a “good” table should satisfy. There are various normal forms (i.e., 1<sup>st</sup> to 6<sup>th</sup>), arranged in increasing order of strictness.

We will be focusing on the third NF (3NF) and the Boyce-Codd NF (BCNF), since they help to get rid of most redundancies, while being always possible to satisfy (unlike the stricter normal forms).

### 8.1 Non-Trivial and Decomposed FD

To simplify our discussions of NF, we introduce the concept of non-trivial and decomposed FD.

- **Decomposed** FD: an FD whose R.H.S. has only one attribute.
  - Note: a non-decomposed FD can always be transformed into an equivalent set of decomposed FDs (i.e., using the *rule of decomposition*).
- **Non-trivial** FD: an FD whose attributes in the R.H.S. do not appear in the L.H.S.

We could derive such FDs using the following algorithm:

1. Given a relation  $\mathcal{R}$ , we consider all attribute subsets in  $\mathcal{R}$ .
2. Compute the closure of each subset.
3. From each closure, remove the *trivial* attributes (i.e., attributes which appear in the L.H.S. of each FD).
4. Derive the non-trivial and decomposed FDs from each closure (using the rule of decomposition).

### 8.2 Boyce-Codd Normal Form (BCNF)

A table  $\mathcal{R}$  is said to be in/satisfy **BCNF** iff every non-trivial and decomposed FD has a superkey as its L.H.S. In other words, if there exists some non-trivial and decomposed FD  $A_1 \dots A_n \rightarrow B$ , then  $A_1 \dots A_n$  must be a superkey in order the table to be in BCNF.

The point of this restriction is to *prevent redundancy*. Suppose  $\exists C_1 \dots C_n \rightarrow B$  where  $C_1 \dots C_n$  is not a superkey. Then, the same  $C_1 \dots C_n$  may appear multiple times in the table, and whenever this happens, the same  $B$  would appear multiple times in the table as well.

We can check whether a table satisfies BCNF using the following algorithm, which makes use of the “*more but not all*” condition:

The table violates BCNF if there exists a closure  $\{A_1 \dots A_k\}$  which (1) contains more than  $k$  attributes, but (2) does not contain all attributes in the table.

1. Compute the closure of each attribute subset in the table  $\mathcal{R}$ .
2. Derive the keys of  $\mathcal{R}$ .

We can compute the closures in the order of increasing attribute subset size

3. For each closure  $\{X_1, \dots, X_k\}^+ = \{Y_1, \dots, Y_m\}$ , check if  $\{X_1, \dots, X_k\}^+$  satisfies the above condition.
4. If such a closure exists, then  $\mathcal{R}$  is not in BCNF.

BCNF has the following properties:

- (+) no update, deletion, or insertion anomalies.
- (+) few redundancies.
- (+) the original table can always be reconstructed from decomposed tables.
- (–) dependencies may not be preserved during decomposition.
  - A decomposition is said to **preserve** all FDs iff  $S$  (i.e., set of FDs on the original table) and  $S'$  (i.e., set of FDs on the decomposed tables) are equivalent.
  - To prove that  $S$  and  $S'$  are equivalent, we will need to show that  $S$  can be derived from  $S'$ , and  $S'$  can be derived from  $S$ .

Dependency preservation makes it easier to avoid “inappropriate” updates.

$S$  is said to be derivable/implied from  $S'$  if all the FDs in  $S$  can be captured by deriving the closures based on the FDs in  $S'$ .

### 8.2.1 BCNF Decomposition

To improve tables which are not in BCNF, we could **decompose**/normalize them into smaller tables. The **recursive binary split algorithm** for BCNF decomposition is described as follows:

1. Find a subset  $X$  of the attributes in  $\mathcal{R}$ , such that its closure  $\{X\}^+$  satisfies the “more but not all” condition.
2. Decompose  $\mathcal{R}$  into two tables  $\mathcal{R}_1$  and  $\mathcal{R}_2$ , such that:
  - $\mathcal{R}_1$  contains all attributes in  $\{X\}^+$ , and
  - $\mathcal{R}_2$  contains all attributes in  $X$ , as well as the attributes which are not in  $\{X\}^+$ .
3. Check if  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are in BCNF.
  - To perform this step, we would need to derive the closures on  $\mathcal{R}$ , and project them onto each  $\mathcal{R}_i$  by *removing irrelevant attributes* (i.e., attributes which are not in the sub-table).
4. Recursively decompose either  $\mathcal{R}_1$  and/or  $\mathcal{R}_2$ , depending on whichever is not in BCNF.

Intuitively, this algorithm works because at each decomposition step, at least one BCNF violation is removed. Since each table can only have a finite number of violations, a recursive decomposition would ensure that all violations will be removed in the end. Note that:



- The BCNF decomposition of a table may not be unique.
- If a table has only 2 attributes, then it *must* be in BCNF.
- BCNF guarantees **lossless join decomposition**.
  - A decomposition is said to be a *lossless join decomposition* iff the common attributes in  $\mathcal{R}_1$  and  $\mathcal{R}_2$  always constitute a superkey of  $\mathcal{R}_1$  or  $\mathcal{R}_2$ .

Hence, when performing decomposition, we could attempt to achieve a binary split at each step which returns a table with only 2 attributes as one of its output.

### 8.3 Third Normal Form (3NF)

Lecture 11  
4<sup>th</sup> April 2022

A table  $\mathcal{R}$  is said to be in/satisfy **3NF** iff for every non-trivial and decomposed FD,

- either the L.H.S. is a superkey, or
- the R.H.S. is a **prime attribute** (i.e., appears in a key of the table).

Notice that 3NF is more lenient than BCNF (i.e.,  $\text{BCNF} \subset \text{3NF}$ ). Thus satisfying BCNF would imply that the table also satisfies 3NF, but not necessarily the other way round.

We can check whether a table satisfies 3NF using the following algorithm:

1. Compute the closure of each attribute subset in the table  $\mathcal{R}$ .
2. Derive the keys of  $\mathcal{R}$ .
3. For each closure  $\{X_1, \dots, X_k\}^+ = \{Y_1, \dots, Y_m\}$ , check if:
  - $\{X_1, \dots, X_k\}^+$  satisfies the “more but not all” condition, and
  - the attribute that is not in  $\{X_1, \dots, X_k\}$  is not a prime attribute.
4. If such a closure exists, then  $\mathcal{R}$  is not in 3NF.

One heuristic to speed up the checking is as follows:

1. Identify the attributes which do not appear in the R.H.S. of any FD (i.e., all keys must contain these attributes).
2. Using these attributes, find the key(s) of the table  $\mathcal{R}$ .
3. Then, we can identify which attributes are not prime attributes.

#### 8.3.1 3NF Decomposition

Before diving into the algorithm, we first introduce the concept of a **minimal basis** of a set  $S$  of FDs, which is a simplified version of  $S$  satisfying the following conditions:

1. Every FD in the minimal basis can be derived from  $S$ , and vice versa.
2. Every FD in the minimal basis is a non-trivial and decomposed FD.
3. No FD in the minimal basis is redundant (i.e., can be derived from the other FDs in the minimal basis).
4. For each FD in the minimal basis, none of the attributes on the L.H.S. is redundant (i.e., if we remove an attribute from the L.H.S., then the FD cannot be derived from  $S$ ).

To find the minimal basis of  $S$ , we could start from the FDs on  $\mathcal{R}$ , and then simplify it step by step according to the algorithm below:

1. Transform the FDs, so that each R.H.S. contains only one attribute.
2. Remove redundant attributes on the L.H.S. of each FD.
  - Whether the removal of an attribute is allowed depends on whether the consequent FD is implied by  $S$ .
3. Remove redundant FDs.

As compared to BCNF, a 3NF decomposition only has one split, which divides the table into two or more parts. The algorithm for 3NF decomposition is given below:

1. Derive a **minimal basis** of  $S$ .
2. In the minimal basis, combine the FDs whose L.H.S. are the same.
3. Create a table for each remaining FD.
4. If none of the tables contain a key of the original table  $\mathcal{R}$ , create a table that contains a key of  $\mathcal{R}$ .
  - The last step helps to ensure a *lossless join decomposition*.

How do we decide between using BCNF and 3NF?

- Typically, we would attempt to satisfy BCNF if we are able to find a BCNF decomposition that preserves all FDs.
- If such a decomposition cannot be found, then:
  - we could go for BCNF if preserving all FDs is not important, and
  - go for 3NF otherwise.

\* \* \*