# CS2102 Cheatsheet

## DBMS Definitions

A *data model* is a collection of concepts for describing data.
A *schema* is a description of the structure of a database using a data model.
A *schema instance* is the content of a database at a particular time.

## Relational Data Model

Data is modelled by *relations*, and each relation has a definition called a *relation schema.*

- Schema specifies attributes and data constraints
- Data constraints include domain constraints

A relation can be seen as a *table with rows and columns*

- Number of columns = *Degree/Arity*
- Number of rows = *Cardinality*

Each row is called a *tuple/record*. It has a *component* for each attribute of the relation.
A relation is thus a set of tuples and an instance of the *relation schema*, i.e. of a single table.
A *domain* is a set of atomic values, e.g. integers. All values for an attribute is either in this domain or *null*.
null is a special value that represents not applicable or unknown.
A *relational database schema* consists of a set of relation schemas and their data constraints, i.e. of multiple tables.
A *relational database* is an instance of the schema and is a collection of tables.

## Integrity Constraints

Condition that restricts the data that can be stored in a database instance.
A *legal relation instance* is a relation that satisfies all specified ICs.

- *Domain constraints* restrict the attribute values of relations, e.g. only integers allowed
- A *key* is a *superkey* which is minimal, i.e. no proper subset of itself is a superkey
  - A *superkey* is a subset of attributes in a relation that unique identifies its tuples
  - Key attribute values cannot be null (*key constraints*)
  - A relation can have multiple keys, called *candidate keys*. One of these keys is selected as the *primary key.*
- A *foreign key* refers to the primary key of a second relation (which can be itself)
  - Each foreign key value must be the primary key value in the referenced relation or be null (*foreign key constraint*)
  - Also known as *referential integrity constraints*

## Relational Algebra

A formal language for asking *queries* on relations.
A *query* is composed of a collection of operators called *relational operators.*
Relations are *closed* under relational operators.

### Selection: $\boldsymbol{\sigma_c}$

$\sigma_c(R)$ selects tuples from relation $R$ that satisfy the selection condition $c$.
Example: $\sigma_{price<20}(Sells)$
The *selection condition* is a boolean combination of *terms.*
A *term* is one of the following forms:
attribute **op** constant; attribute$_1$ **op** attribute$_2$; term$_1$ **and** term$_2$; term$_1$ **or** term$_2$; **not** term$_1$; (term$_1$)

- op $\in \{=, <>, <, \leq, >, \geq\}$.
- Operator precedence: (), op, not, and, or.

Result of a *comparison operation* involving a null value is *unknown.*
Result of an *arithmetic operation* involving a null value is *null.*
A tuple is only selected if the condition evaluates to *true* on it.

| 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 | |

Table: Three-valued logic system

### Projection: $\boldsymbol{\pi_l}$

$\pi_l(R)$ projects attributes given by a list $l$ of attributes from relation $R$.
Example: $\sigma_{rname,pizza}(Sells)$
Duplicate records are removed in the output relation.

### Renaming: $\boldsymbol{\rho_l}$

$l$ is a list of attribute renamings of the form $a_1{:}\,b_1, \cdots, a_n{:}\,b_n$.
The order of the attribute renamings in $l$ does not matter.

### Set Operators

*Union:* $R \cup S$ returns a relation containing all tuples that occur in $R$ or $S$
*Intersection:* $R \cap S$ returns a relation containing all tuples that occur in both $R$ and $S$
*Set-difference:* $R - S$ returns a relation containing all tuples in $R$ but not in $S$
These operators require input relations to be *union compatible*

- Same number of attributes, and
- Corresponding attributes have the same domains
- No need to use the same attribute names

The resultant schema will follow the schema of $R$ above.

### Cross-Product: $\times$

Given $R(A, B, C)$ and $S(X, Y)$, $R \times S$ gives us $(A, B, C, X, Y)$

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

Also known as *cartesian product*

- When finding pairs of the same type, e.g. pairs of customers, we can add an additional selection of e.g. $C_1 < C_2$ to filter out duplicates.

### Inner Join: $\boldsymbol{R \bowtie_c S}$

$$R \bowtie_c S = \sigma_c(R \times S)$$

### Natural Join: $\boldsymbol{R \bowtie S}$

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

where

- $A = \{a_1, a_2, \cdots, a_n\}$ is the set of common attributes between $R$ and $S$
- $c = (a_1 = b_1)$ and $\cdots$ and $(a_n = b_n)$
- $l$ includes, in this order
  - List of attributes in $R$ that are also in $A$
  - List of attributes in $R$ that are not in $A$
  - List of attributes in $S$ that are not in $A$

### Dangling Tuples: $\boldsymbol{dangle(R \bowtie_c S)}$

Let $attr(R)$ be the list of attributes in the schema of $R$.
We say that $t \in R$ is a dangling tuple in $R$ wrt $R \bowtie_c S$ if $t \notin \pi_{attr(R)}(R \bowtie_c S)$

### Left Outer Join: $\boldsymbol{R \rightarrow_c S}$

Let $null(R)$ denote a tuple of null values of same arity as $R$

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

### Right Outer Join: $\boldsymbol{R \leftarrow_c S}$

$$R \leftarrow_c S = (R \bowtie_c S) \cup \left(\{null(R)\} \times dangle(S \bowtie_c R)\right)$$

### Full Outer Join: $\boldsymbol{R \leftrightarrow_c S}$

$$R \leftrightarrow_c S = (R \rightarrow_c S) \cup \left(\{null(R)\} \times dangle(S \bowtie_c R)\right)$$

### Natural Left Outer Join: $\boldsymbol{R \rightarrow S}$

Same schema as $R \bowtie S$

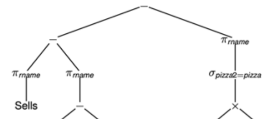
### Natural Right Outer Join: $\boldsymbol{R \leftarrow S}$

Same schema as $R \bowtie S$

### Natural Full Outer Join: $\boldsymbol{R \leftrightarrow S}$

Same schema as $R \bowtie S$

### Operator Trees




## Structured Query Language (SQL)

Domain-specific language designed for computations on relations. It is a declarative language. Contains two parts: *Data Definition Language* and *Data Manipulation Language*

### Data Types

- boolean: true/false (null = unknown)
- integer: signed four byte integer
- float8: double-precision floating point number (8 bytes)
- 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 up to n characters
- text: variable-length character string
- date: calendar date (year, month, day)
- timestamp: date and time

### Create/Drop Table

```
create table Students (
  studentId    integer,
  name         varchar(100),
  birthDate    date
  dept         varchar(20) default 'CS'

);
drop table Students;
drop table if exists Students;
// deletes objects that depend on this table
drop table if exists Students cascade;
```

### Is Null Predicate

```
x IS NULL
```

It evaluates to true for null values, else false

```
x IS NOT NULL = NOT (x IS NULL)
```

**Is Distinct From Predicate**

`x IS DISTINCT FROM y`

Equivalent to `x <> y` if both `x` and `y` are non-null, else is false if both are null, else true if only one is null

`x IS NOT DISTINCT FROM y = NOT(x is DISTINCT FROM y)`

**not null Constraint**

```
name        varchar(100) not null,
```

**unique Constraint**

```
studentId   integer unique,
```

or

```
unique (city, state) -- at bottom
```

**primary key Constraint**

```
studentId   integer primary key,
```

or

```
studentId   integer unique not null,
```

or

```
primary key (sid, cid) -- at bottom
```

**foreign key Constraint**

```
studentId   integer references Student (id),
```

or

```
foreign key (a, b) references Other (a, b) -- at bottom
```

If any of the referencing columns is null, a referencing row can escape satisfying the foreign key constraint. If we add `match full` to the end of the constraint, then a referencing row only escapes if all of the referencing columns are null.

**check Constraints**

```
check (day in (1,2,3,4,5)), -- besides day
check ((hour >= 8) and (hour <= 17)), -- at bottom
```

**Constraint Names**

```
bDate   date constraint bdate check (bdate is not null)
```

or

```
constraint obj_pri_key primary key (name,day,hour)
```

**Insert**

```
insert into Students
values (12345, 'Alice', '1999-12-25', 'Maths');
insert into Students (name, studentId)
values ('Bob', 67890), ('Carol', 11122);
```

**Delete**

```
delete from Students; -- deletes all
delete from Students where dept = 'Maths';
```

**Update**

```
update Accounts set balance = balance * 1.02;
update Accounts set  balance = balance + 500, name = 'Alice'
where accountId = 12345;
```

**Foreign Key Constraints Violations**

*No Action*: Rejects action if it violates constraint (default)
*Restrict:* Same as No Action but constraint checking is not deferred
*Cascade*: Propagates delete/update to referencing tuples
*Set Null:* Updates foreign keys to null
*Set Default:* Updates foreign keys to some default value. We will need to specify this value at the referencing column, and it must meet the foreign key constraints, else action will fail

```
foreign key (a, b) references Other (a, b)
            on delete cascade
```

**Transactions**

Starts with `begin;` and ends with `commit;` or `rollback;`. Can contain multiple SQL statements.

- *Atomicity*: Either all effects are reflected or none.
- *Consistency*: Executed in isolation, preserves the DB consistency
- *Isolation*: From the effects of other concurrent transactions
- *Durability*: Effects persist even if system failures occur

**Deferrable Constraints**

unique, primary key and foreign key constraints can be deferred using `deferrable initially deferred` or `deferrable initially immediate`

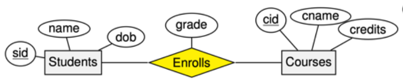
We can change this using `set constraints`, e.g. `set constraints fkey deferred;` or `set constraints fkey immediate;` (retroactive)

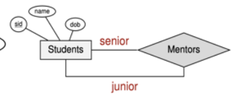
**Modifying Schema**

```
alter table Students alter column dept drop default;
alter table Students drop column dept;
alter table Students add column faculty varchar(20);
alter table Students add constraint fk_grade foreign key
(grade) references Grades;
```

**Entity-Relationship (ER) Model**

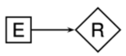
*Entity*: Real-world object distinguishable from other objects
*Attribute*: Specific information describing an entity, represented by ovals
*Entity set*: Collection of similar entities, represented by rectangles
*Key*: Represented as underlined attributes
*Relationship*: Association among two or more entities
*Relationship set*: Collection of similar relationships, represented by diamonds. Attributes are used to describe information about relationships.



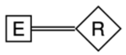




By default, these relationships are *many-to-many*.
*Relationship role*: Shown explicitly when one entity set appears two or more times in a relationship set
*Degree*: An $n$-ary relationship set involves $n$ entity roles; degree $= n$.
$n = 2 \rightarrow$ binary, $n = 3 \rightarrow$ ternary.
*Relationship keys*: Each relationship set instance will have the primary keys of the entities as well as its own attributes. The primary key of the relationship set will contain those primary keys as well as a subset of its own attributes, which will be underlined.

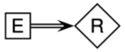
**Relationship Constraints**




*Key Constraint*: Each instance of $E$ can participate in *at most one* instance of $R$. Represented by an arrow. Allows for *one-to-many* if one entity has an constraint but the other doesn't, or *one-to-one* if both entities have the constraint.




*Total Participation Constraint*: Each instance of $E$ must participate in *at least one* instance of $R$. Represented by a double line.
A single line is a *partial participation constraint*, i.e. 0 or more.



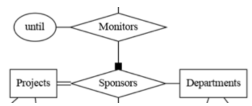

*Key & Total Participation Constraint:* Each instance of $E$ participates in exactly one instance of $R$. Represented by double line arrow.

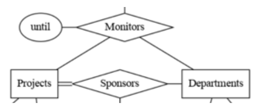

*Weak Entity Set*: $E$ is a weak entity set with identifying owner $E'$ & identifying relationship set $R$. $E$ does not have its own key and requires the primary key of its *owner entity* to be uniquely identified. It must have a many-to-one relationship with $E'$ **and** total participation in $R$.
*Partial Key:* Set of attributes of a weak entity set that uniquely identifies a weak entity for a given owner entity.

**Database Design**

We can represent primary key. Cannot represent unique, not null.
Without constraints, we generally represent relationship sets as tables with foreign keys that form part of its primary key, i.e. association class.
With constraints, we can choose to represent them as a separate table or as an attribute in an entity set's table (the one that's many in many-to-one, or any in an one-to-one).
Note that using a separate table for key & total participation constraint will not enforce that constraint by schema, while the latter method does.
If the same entity set participates twice in a relationship (i.e. relationship roles), we can have two self-referencing foreign keys, `firstId` and `secondId`, in the table and a `check (firstId <> secondId)`.
Weak entity set & its identifying relationship set can be represented as a single relation, which has a foreign key that deletes on cascade and forms part of its primary key.

**Aggregation**

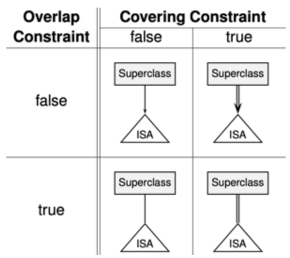



When a relation between two entities is treated as a single entity. The above two diagrams are not the same, since the right does not enforce a relationship between `Projects` and `Departments`.

**ISA Hierarchies**

We can classify an entity set into subclasses. Every entity in a subclass entity set is an entity in its superclass entity set




- *Overlap Constraint*: Can an entity belong to multiple subclasses?
- *Covering Constraint*: Does an entity in a superclass have to belong to some subclass?

We can create a relation per subclass/superclass, with the subclass' tables having a foreign key referencing the superclass' table, along with additional attributes.

# CS2102 Cheatsheet

## General

A *key* is a minimal *superkey*, i.e. no proper subset of itself is a superkey

- A *superkey* is a subset of attributes that unique identifies its tuples
- Key attribute values cannot be null (*key constraints*)
- Can have multiple keys (*candidate keys*). One is the *primary key*.

A *foreign key* refers to primary key of a second relation (can be itself)

- Each foreign key value must be the primary key value in the referenced relation or be null (*foreign key constraint*)

## SQL Data Types

boolean, integer, float8, numeric, numeric(p, s), char(n), varchar(n), text, date, timestamp

## Create/Drop Table

```
create table Students (
  dept        varchar(20) default 'CS'
);
drop table if exists Students cascade;
```

**Is Null Predicate:** `x IS NULL`

**Is Distinct From Predicate:** `x IS DISTINCT FROM y`

Equivalent to `x <> y` if both `x` and `y` are non-null, else is false if both are null, else true if only one is null

**not null Constraint:** `name varchar(100) not null,`

**unique Constraint:** `studentId integer unique,` or `unique (city, state) -- at bottom`

**primary key Constraint:** `studentId integer primary key,` or `primary key (sid, cid) -- at bottom`

**foreign key Constraint:** `studentId integer references Student (id),` or `foreign key (a, b) references Other (a, b) -- at bottom`

If any of the referencing columns is null, a referencing row can escape satisfying the FK constraint. If we add `match full` to the end of constraint, then a referencing row only escapes if all of the referencing columns are null.

## check Constraints

```
check (day in (1,2,3,4,5)), -- besides day
check ((hour >= 8) and (hour <= 17)), -- at bottom
```

## Constraint Names

`bDate   date constraint bdate check (bdate is not null)` or `constraint obj_pri_key primary key (name, day, hour)`

**Insert:** `insert into Students (name, studentId) values ('Bob', 67890), ('Carol', 11122);`

**Delete:** `delete from Students where dept = 'Maths';`

**Update**: `update Accounts set  balance = balance + 500, name = 'Alice' where accountId = 12345;`

## Foreign Key Constraints Violations

*No Action*: Rejects action if it violates constraint (default). *Restrict:* Same as No Action but constraint checking is not deferred. *Cascade*: Propagates delete/update to referencing tuples. *Set Null:* Updates foreign keys to null. *Set Default:* Updates foreign keys to some default value. We will need to specify this value at the referencing column, and it must meet the foreign key constraints, else action will fail

`foreign key (a, b) references Other (a, b) on delete cascade`

## Transactions

Starts with `begin;` and ends with `commit;` or `rollback;`. Can contain multiple SQL statements.

*Atomicity*: Either all effects are reflected or none. *Consistency*: Executed in isolation, preserves DB consistency. *Isolation*: From the effects of other concurrent transactions. *Durability*: Effects persist even if system fails.

## Deferrable Constraints

unique, primary key and foreign key constraints can be deferred using `deferrable initially deferred` or `deferrable initially immediate`

We can change this using `set constraints`, e.g. `set constraints fkey deferred;` or `set constraints fkey immediate;` (retroactive)

## Modifying Schema

```
alter table Students alter column dept drop default;
alter table Students drop column dept;
alter table Students add column faculty varchar(20);
alter table Students add constraint fk_grade foreign key (grade) references Grades;
```

## Expressions in Select Clause

- `distinct`: Removes duplicates
- `A as B`: Rename output column `A` to `B`, column alias
- `'String' || value`: String concatenation
- `round(A)`: Rounds to nearest integer value

## Set Operations

`union`, `intersect` and `except` are same as $\cup, \cap, -$. Eliminates duplicates.

`union all`, `intersect all` and `except all` preserve duplicate records.

Example: `select A from R except select B from S;`

## Multi-Relation Queries

`Rfull as R, Sfull as S` or `Rfull R, Sfull S`: Cross-product

`R inner join S on R.A = S.A`, `R join S on R.A = S.A`

`R natural join S`

`R left outer join S on R.A = S.A`, `R left join S on R.A = S.A`

`R natural left outer join S`, `R natural left join S`

## Subqueries

`exists`: Returns true if subquery is non-empty. E.g. `where exists`

`in`: Subquery must return a single column, and true if it is non-empty and one row equals expression. E.g. `where expression in`

`any`: Same as in but instead of equality, checks for result of binary operation. E.g. `where price > any`

`all`: Same as any but requires all to match, e.g. `where price >= all`

- To use the above with multiple columns, use row constructors, e.g. `where row(day, hour) <= all` or `where (day, hour) <= all`.
- Can be used in `where`, `from` and `having`.
- If used in `from`, it must be enclosed in parentheses and assigned table alias. E.g. `select * from (subquery) as persons(name, age);`

## Scalar Subqueries

A subquery that returns at most one tuple with one column. If empty, then `null` is returned. It can be then used as a scalar expression. Also called **nested queries** or an **inner query** nested within an **outer query**.

In terms of table alias (tuple variable) scoping rules, it's the same as a normal language. If the inner query refers to a tuple variable declared in an outer query, then it is a **correlated nested query**.

**Order By:** `order by area asc, price desc; order by area, price desc;`

**Limit:** `limit 3`: Show top 3.

**Offset:** `offset 3`: Show $4^{\text{th}}$ and onwards.

## Aggregate Functions

`min(A)`, `max(A)`, `avg(A)`, `sum(A)`, `count(A)` (non-null), `count(*)` (all rows, may be null), `avg(distinct A)` (non-null), `sum(distinct A)` (non-null), `count(distinct A)` (non-null).

- Performed on an empty relation OR a column entirely of `null`s, all functions except `count` will return `null`. `count` will return `0`, unless `count(*)` is used, in which case `n` will be returned.
- Can be used in `select`, `having` and `order by`.
- If a `select` clause contains an aggregated function and there is no `group by` clause, then the `select` clause must not contain any column that is not in an aggregated expression, i.e. all or nothing.

## Group By

Conditions for a `group by` clause on relation `R`:

- Output column `A` must appear in the `group by` clause, OR
- `A` appears in an aggregated expression in the `select` clause.
- The primary key of `R` appears in the `group by` clause.

## Having

This is similar to `where`, but is applied after `group by`.

Conditions for a `having` clause on relation `R`:

- Column `A` that appears in the `having` clause must appear in the `group by` clause, OR
- `A` appears in an aggregated expression in the `having` clause.
- The primary key of `R` appears in the `group by` clause.

## Conceptual Evaluation of Queries

```
select          distinct select-list
from            from-list
where           where-condition
group by        groupby-list
having          having-condition
order by        orderby-list
offset          offset-specification
limit           limit-specification
```

1. Compute cross-product of tables in from-list
2. Select tuples that evaluate to true for the where-condition
3. Partition selected tuples into groups using groupby-list
4. Select the groups that evaluate to true for the having-condition
5. For each selected group, generate an output tuple based on select-list
6. Remove duplicate output tuples because of distinct
7. Sort the output tuples based on orderby-list
8. Remove the appropriate output tuples based on offset-specification and limit-specification

## Common Table Expressions

We can declare common table expressions shared by all subsequent queries.

```
with
R1 as (Q1),      -- comma for non-last CTEs
R2 as (Q2)       -- no comma for final CTE
```

## Views

Virtual relation that can be used for querying.

`create or replace view v3 (rname1, rname2) as`

This creates an interface, where users/applications query using the schema of the view, thus safe from changes to the actual logical/physical schema.

## Case

```
case
    when marks >= 70 then 'A'
    else 'D'
end as grade
case grade                 -- case expression
  when 'A' then 70
  else 0
end as marks
```

**Coalesce:** `select coalesce(third, second, first) as result`
Returns the first non-`null` value in its arguments.
**Nullif:** `select name, nullif(result, 'absent') as status`
Returns `null` if result is equal to 'absent', otherwise result.
**Like / Similar To:** `where cname like '___%e';`
Underscore matches a single character

- % matches any sequence of 0 or more characters
- More complex regex will need `similar to`.

**Functions in SQL**

```
create or replace function a(x int) returns char(1) as $$
    -- select a single character using x
$$ language sql;
returns R
returns setof R
d(out x int, out y int) returns record as $$ -- called using
SELECT d(); returns single tuple
e(out x int, out y int) returns setof record as $$
f() returns table(x int, y int) as $$ -- same as e
```

You can use `inout` for the parameters if the input "schema" and the output "schema" share columns.

**Procedures**

```
create or replace procedure g(x int) as $$ -- CALL g(x);
```

**Variables and Control Structures**

```
as $$
declare
  temp_val integer;
begin
  -- function body
end;
$$ language plpgsql;

loop
  exit when condition;
  statements;
end loop;
```

```
if condition1 then
  statement1;
elsif condition2 then
    statement2;
else
  else-statement;
end if;
```

**Cursor**

```
declare
  curs cursor for (select * from R order by A desc);
  r record;
begin
  open curs;
  loop
      fetch curs into r;
      exit when not found;
      -- assign output table column names to values
      return next;
  end loop;
  close curs;
end;
fetch prior from cur into r; fetch first from cur into r;
fetch last from cur into r; fetch absolute 3 from cur into r;
```

**Triggers**

```
create or replace function func() returns trigger as $$
create trigger trigger_name [before / after]
[insert / update /delete] on R for each [row / statement]
execute function func();
```

Order of triggers: before statement, before row, after row, after statement

**Special Values in Triggers**
`TG_OP`: Operation that activates the trigger, i.e. `'INSERT'`, `'UPDATE'`, `'DELETE'`, `'TRUNCATE'`
`TG_TABLE_NAME`: Name of table causing the invocation
`OLD`: Old tuple being updated or deleted. `null` for insertion.
`NEW`: New tuple being inserted or updated. `null` for deletion.

**Instead Of Triggers (Only Row-Level)**
You can also define `instead of [insert / update / delete]` triggers for views (only), e.g. to update the actual tables instead.

**Return Values for Row-Level Triggers**
`before insert`: non-null t – t will be inserted, null – no insertion
`before update`: non-null t – t will be the updated tuple, null – no update
`before delete`: non-null t – deletion happens, null – no deletion
`after [insert/update/delete]`: return value does not matter
`instead of`: non-null t – proceed, null – ignore operations on current row
If a `before` row-level trigger returns null, then all subsequent triggers on the same row are omitted.

**Return Values for Statement-Level Triggers**
Return values are ignored. Raise exceptions if to ignore operations.

**Trigger Condition**
Example: `for each row when (NEW.Name = 'Hello') execute`
No `select` in `when`, no `OLD` in `when` for `insert`, no `NEW` in `when` for `delete`, no `when` for `instead of`.

**Deferred Triggers**
`create constraint trigger trigger_name after … on R deferrable initially [deferred / immediate] for each row …`
Only works for `after` and `for each row`.
If `initially immediate`, we need to change on the fly
`begin transaction; set constraints trigger_name deferred;`

**Functional Dependencies**
A → B means A decides B, i.e. if two rows have the same A, then they have the same B.

Armstrong's Axioms

- Reflexivity: ABC → A (set to subset)
- Augmentation: If A → B, then AC → BC for any C
- Transitivity: If A → B and B → C then A → C

Extended Axioms

- Decomposition: If A → BC then A → B and A → C
- Union: If A → B and A → C then A → BC

**Closures**
$\{A\}^+$ denotes the set of attributes that can be directly or indirectly decided by A, also called the closure of A.
To compute: start with {A}; for all FDs such that the LHS can be found in the current closure, put the RHS into the closure as well; repeat until no more new attributes can be added.
To prove X → Y, we just need to show $\{X\}^+$ contains Y. Opposite is true.

**Keys, Superkeys and Prime Attributes**
*Superkey:* A set of attributes in a table that decides all other attributes. *Key:* A superkey that is minimal. *Prime attribute:* Attribute that appears in a key.
To find keys of T: consider all subsets of T; derive the closure of each subset; identify the superkeys; identify the keys.
Heuristic: Check smaller attribute sets first.
Heuristic: If an attribute does not appear in the RHS of any FD, then it must be in every key.

**Non-Trivial and Decomposed FDs (NTD)**
*Decomposed FD:* RHS only has one attribute. *Non-trivial and decomposed:* RHS does not appear in LHS.
To find such FDs, simply compute all closures, remove trivial attributes, then separate them into decomposed FDs.

**Boyce-Codd Normal Form (BCNF)**
*Normal form:* Definition of minimum requirements in terms of redundancy.
A table R is in BCNF if every NTD has a **superkey** as its LHS.
To check if R is in BCNF: compute all closures; find the keys from the closures; derive NTD FDs from the closures; check the BCNF requirement.
Heuristic: Check if there's a closure that contains "more but not all", i.e. the RHS has more attributes than the LHS, but does not contain all attributes.

**BCNF Decomposition**
Find any "more but not all" closure $\{X\}^+$, then decompose R into $R_1$ and $R_2$:

- $R_1$ contains all attributes in $\{X\}^+$ (i.e. RHS) and
- $R_2$ contains all attributes in X and attributes not in $\{X\}^+$ (i.e. LHS + remaining attributes).

Then, project the closures from R onto $R_1$ or $R_2$, then repeat the process.
To project: enumerate all attribute subsets in $R_1$ (WLOG); derive their closures on R; remove irrelevant attributes.
If a table only has two attributes, then it must be in BCNF.

**BCNF Properties**
No update or deletion or insertion anomalies, small redundancy and you can always reconstruct the original table.
A **lossless join** is guaranteed whenever the common attributes in $R_1$ and $R_2$ constitute a superkey of $R_1$ or $R_2$.
But it may not guarantee **dependency preservation**.

**Dependency Preservation**
Let S be the given set of FDs on the original table, and S' be the set of FDs on the decomposed tables.
A decomposition preserves all FDs iff S and S' are equivalent, i.e. every FD in S' can be derived from S, and vice versa.
Preserving FDs allow us to prevent inappropriate updates.

**Third Normal Form (3NF)**
A table satisfies 3NF iff for every NTD, either the LHS is a superkey or the RHS is a prime attribute.
This is more lenient than BCNF, i.e. satisfying BCNF → satisfying 3NF.
The checking is similar to BCNF, except that the requirement is different.

**3NF Decomposition**
3NF decomposition does a single split into two or more parts.
Derive the minimal basis of the FDs; combine the FDs whose LHS are the same, i.e. A → B and A → C becomes A → BC; create a table for each remaining FD, e.g. R1(A, B, C); if none of the tables contain a key of the original table R, create a table that contains any key of R.

**Minimal Basis / Cover (MB)**
We simplify a set S of FDs using the four conditions:

1. Every FD in the MB can be derived from S and vice versa.
2. Every FD in the MB is a NTD.
3. No FD is redundant, i.e. can be derived from other FDs in the MB.
4. For each FD, none of the LHS attributes is redundant, i.e. if we remove it, it cannot be derived from the original set of FDs.

Algorithm for MB
Transform the FDs so that each RHS only contains one attribute; remove redundant attributes on the LHS of each FD; remove redundant FDs.