Database Management System

Transactions

defined as a **finite sequence** of database operations *read/write* and constitues the smallest logical unit of work from the application perspective.

Properties of each transaction, T

- Atomitricty - all/none of T reflected in the database
- Consistency - T guarantees the correct state
- Isolation - T isolated from effects of concurrent transactions
- Durability - after T, effects are permanent

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.

Relational Data Model

A relational database schema is a set of relation schemas + data constraints. Data is modelled by *relations*, each *relation* defined as a **relation schema**:

- attributes: columns
- data constraints: domain constraints → datatype
- `Employees(id: INT, name: TEXT, salary: NUMERIC())`

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

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

A **domain** is a set of atomic values (INT, NUMERIC, TEXT) denoted as ***dom***(A_i). A **relation** is a set of tuples (records in the database), denoted as $\boldsymbol{R}(A_1, A_2, ..., A_n)$.
• every v of attribute A_i is either: v $\in$ $dom(A_i)$ / v = NULL.
• we write $\boldsymbol{R.A}_i$ to refer to the attribute A_i of relation R.

Data Integrity

defined as conditions that restrict what constitutes valid data.

Keys

- **superkey**: subset of attributes that uniquely identifies a tuple in a relation
- **key**: is a ***superkey*** that is also minimal → no proper subset of the key is a superkey.
  • minimal (cannot be made smaller) ≠ minimum (smallest)
- **candidate keys**: set of all **keys** of a given relation
- **primary key**: selected **candidate keys**, and they CANNOT be NULL. To simplify our notation, we underline our primary keys in the schema notation.

Foreign keys

- subset of attributes of relation R_1 that refers to the **PK** of relation R_2. → R_1: referencing, R_2: referenced relation
- **requirements**: each FK in R_1 must either appear as a PK in R_2 **OR** be a NULL value (in a tuple, containing at least 1 NULL value)

Relational Algebra

Closure Property

A set of values is **closed** under the set of operators if any combinations of the operators produces values in the given set
• Relations *closed* under RA, we can chain operations

Three-Valued Logic

Logical Operations

Truth Table

c_1	c_2	$c_1 \wedge c_2$	$c_1 \vee c_2$	$\neg c_1$
False	False	False	False	True
False	NULL	False	NULL	True
False	True	False	True	True
NULL	False	False	NULL	NULL
NULL	NULL	NULL	NULL	NULL
NULL	True	NULL	True	NULL
True	False	False	True	False
True	NULL	NULL	True	False
True	True	True	True	False

Basic Operators

Selection: σ_c

$\sigma_c(R)$ selects tuples from relation R, satisfying the selection condition c. e.g.: $\sigma_{price<20}(Sells)$. The *selection condition* is a boolean condition 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.
- A tuple is only selected if condition evalues to *true* on it.

Projection ρ_l

l is a list of attribute renamings of the form $a_1{:}b_1,\ \dots,\ a_n{:}b_n$. The order of the attribute renamings in l does not matter, however it will remove any duplicates after projection.

Set Operators

The attributes require input relations to be ***union compatible***:

Union: $R \cup S$ returns a relation containing all tuples in R or S
Intersection: $R \cap S$ returns a relation containing all tuples in both R and S
Set-difference: R - S returns a relation containing all tuples in R but not in S.

Union Compatability:

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

Cross-Product: $\times$

Given $R(A, B, C)$ and $S(X, Y)$, $R \times S = (A, B, C, X, Y)$, where every combination of A - Y is an entry in the new relation. $\star$ The set of attributes in R and S must be ***disjoint***.

Inner Join

Include only tuples that satisfy the condition.

θ-join: $\boldsymbol{R} \bowtie_\theta \boldsymbol{S}$

$$R \bowtie_\theta S = \sigma_\theta(R \times S)$$

Equi Join: $\boldsymbol{R} \bowtie_= \boldsymbol{S}$

Special type of θ-join where the only relational operator that can be used is equality.

$$R \bowtie_= S = \sigma_=(R \times S)$$

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

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

where

- A = { a_1, a_2, ..., a_n } is the set of common attributes between R and S
- c = (a_1 = b_1) and ... 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

Outer Join

Include only tuples that do not satisfy the condition.

Dangling Tuple

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 $R \bowtie S$ if $t \notin \pi_{attr(R)}(R \bowtie_c S))$

Left Outer Join: $\boldsymbol{R}$ ⟕$_{[\theta]}$ $\boldsymbol{S}$

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

$$R ⟕_{[\theta]} S = R \bowtie_{[\theta]} S \cup (dangle(R \bowtie_{[\theta]} S) \times \{null(S)\})$$

Right Outer Join: $\boldsymbol{R}$ ⟖$_{[\theta]}$ $\boldsymbol{S}$

$$R ⟖_{[\theta]} S = R \bowtie_{[\theta]} S \cup (dangle(\{null(S)\} \times R \bowtie_{[\theta]} S))$$

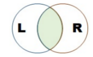
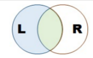
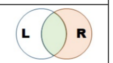
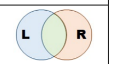
Full Outer Join: $\boldsymbol{R}$ ⟗$_{[\theta]}$ $\boldsymbol{S}$

$$R ⟗_{[\theta]} S = R \bowtie_{[\theta]} S \cup (dangle(R \bowtie_{[\theta]} S) \times \{null(S)\}) \cup (dangle(\{null(S)\} \times R \bowtie_{[\theta]} S))$$

Natural Left/Right/Outer Joins

Same schema as $R \bowtie S$

Joins

L	Operator	R	Visualization
	$\bowtie$		
Keep dangling	⟕		
	⟖	Keep dangling	
Keep dangling	⟗	Keep dangling	

Complex Equations - Equivalence

We say that 2 relational algebra expressions Q_1 and Q_2 are equivalent if for any input relations either: both produces error **OR** both produces the same result. → **strongly** equivalent.
⋆ **Weakly** equivalent can be defined as if there are no error then both produces the same result.

SQL - Structured Query Language

Domain-specific language designed for computations on relations. Made of: *Data definition*, *Data manipulation*, *Data Query*, *Data Control* and *Transaction Control*.

Data Types

- **boolean:** `true/false` (`null` == unknown)
- **integer:** signed 4 byte integer
- **float8:** double-precision floating point number (8 bytes)
- **numeric:** arbitrary precision floating point number
- **numeric(p, s):** max total of p digits with max 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:** calender date (year, month, day)
- **timestamp:** date and time

Extended Data Types

- **Document:** `XML/JSON`
- **Spatial:** Point/Line/Polygon/Circle/Box/Path
- **Special:** Money/Currency & MAC/IP address

Create/Drop Table

```
CREATE TABLE Employees (
    id INTEGER,
    name VARCHAR(50),
    age INTEGER,
    role VARCHAR(50) DEFAULT 'sales'
);
```

Insertion

```
INSERT INTO Employees
  VALUES (101, 'Sarah', 25, 'dev');
INSERT INTO Employees (name, id)
  VALUES ('Judy', 102), ('Max', 103);
```

- Either **all** inserted or **none** inserted
- Attributes can be specied out-of-order (optionally)
- Missing values are replaced with NULL (if allowed) or default values (if specied)

Deleting

```
DELETE FROM <table_name>
  [ WHERE <condition> ];
```

- Condition is optional, unspecified: equivalent to always true
- Specified, condition can be arbitrarily complex

Integrity Constraints

Principle of Acceptance: Perform the operation if the condition **evaluates to TRUE**.
Principle of Rejection: Reject the insertion if the condition **evaluates to FALSE**.

Type	Column	Table	Condition
Not-NULL	`NOT NULL`	-	`IS NOT NULL`
Unique	`UNIQUE`	`UNIQUE(`$A_1, A_2, \ldots$`)`	`x.`A_i `<>` `y.`A_i
Primary Key	`PRIMARY KEY`	`PRIMARY KEY(`$A_1, A_2, \ldots$`)`	`UNIQUE` & `NOT NULL`
Foreign Key	`REFERENCES` R_1`(B)`	`FOREIGN KEY (`$A_1, A_2, \ldots$`)` `REFERENCES` R_1`(`$B_1, B_2, \ldots$`)`	The tuple exists in R_1 or the tuple contains `NULL` value
General	`CHECK (c)`	`CHECK (c)`	Condition `c` does not evaluate to `False`

Is Null Predicate

`x IS NULL`: 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 false if both are null, else true if only one is null
`x IS NOT DISTINCT FROM y = (x IS DISTINCT FROM y)`

non-null Constraint

`name      varchar(100) NOT NULL,`

unique Constraint

`studentId      INT UNIQUE`, or
`unique (city, state) -- at bottom` or
`primary key (sid, cid) -- at bottom`

primary key Constraint

Primary key is a selected candidate key → uniquely identifies a tuple in a relation and cannot be NULL, `UNIQUE` and `NOT NULL`
`studentId      INT CONSTRAINT stdnt_pk PRIMARY KEY` or
`PRIMARY KEY (eid, pname), -- at bottom`

foreign key Constraint

`studentId      INT REFERENCES Student (id)` or
`FOREIGN KEY (a, b) REFERENCES Other (a, b) -- at bottom`

Keyword	Action
`NO ACTION`	*Reject* delete/update if it violates constraint *(default value)*
`RESTRICT`	Similar to "`NO ACTION`" except that check of constraint *cannot be deferred* *(deferrable constraints are discussed in a bit)*
`CASCADE`	*Propagates* delete/update to the referencing tuples
`SET DEFAULT`	*Updates* the foreign key of the referencing tuples to some default value *(**Important:** default value must be a primary key in the referencing table)*
`SET NULL`	*Updates* the foreign key of the referencing tuples to `NULL` value *(**Important:** corresponding column must be allowed to contain `NULL` values)*

Modifying Database

Alter Table

Common changes: Adding/dropping columns, constraints or Changing specification of a column (data type, default values)

```
ALTER TABLE <table_name>
  [ALTER / ADD / DROP] [COLUMN / CONSTRAINT] <name>
  <changes>;
```

Drop Table

```
DROP TABLE
[IF EXISTS] // no error if table does not exist
<table_name>[, <table_name> [, <table_name> [...]]]
[CASCADE]; // also delete referencing table
```

Defferable Constraints

unique, *primary key* and *foreign key* constraints can be deffered using:

- `DEFERRABLE INITIALLY DEFFERED`: checked only at the end of the transaction.
- `DEFERRABLE INITIALLY IMMEDIATE`: checked after each statement.
- `NOT DEFFERABLE`: default for constraints

Considerations for defferable constraints

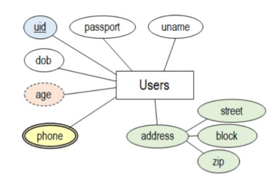
(+)	(-)
• No need to care about the order of SQL statements within a transaction – allow intermediate state to temporarily violate constraints • Allows for cyclic foreign key constraints • Performance boost when constraint checks are bottlenecked – batch insert of larger number of tuples	• Troubleshooting can be more difficult • Data definition no longer unambiguous • May incur performance penalty when performing queries → Certain checks may need to be done at run-time especially during the time when the constraint check is deferred

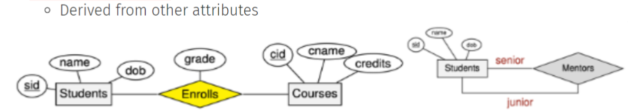
Entity-Relationship (ER) Model

Entity: Real-world object distinguishable from other objects
Attribute: Specific information describing an entity, ovals
Entity set: Collection of similar entities, rectangles
Key: Represented as underlined attributes
Relationship: Association among 2 or more entries *Relationship set*: Collection of similar relationships, represented by diamonds. Attributes used to describe information about relationships.

Subtypes:

1. **Key Attributes** : *underlined*
   - Uniquely identifies each entity
2. **Composite Attributes** : *composed of other ovals*
   - Composed of multiple other attributes
3. **Multivalued Attributes** : *double-lined*
   - One or more values for a given entity
4. **Derived Attributes** : *dashed line*
   - Derived from other attributes

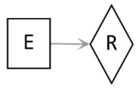




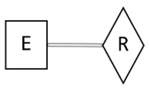
By default these relationships are *many-to-many*.

Relationship role: Shown explicitly when 1 entity appears 2 or more times in a relationship set
Degree: An n-ary relationship set involves n entity roles; degree = n. (n = 2 → binary, n = 3 → ternary) *Relationship keys*: Each relationship set instance will have the primary keys of the entities as well as its own attributes. The primary keys of the relationship set will contain those primary keys as well as a subset of its own attributes, which will be underlined.

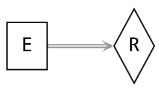
Relationship Constraints



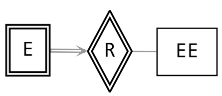
Key Constraint: Each instance of E can participate in **at most one instance** of R. Represented by an arrow. Allows for one-to-many if one entity has a constraint but the other does not, or one-to-one if both entities have the constraint.



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 excatly one instance of R. Represented by double line arrow.



Weak Entity Set: E is a weak entity set with identifying owner EE & 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 EE **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.

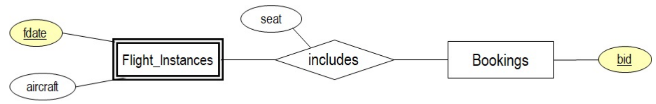
Relational Mapping

ER Diagram	Schema
Name of entity set	Name of table
Attribute of entity set	Attribute/column of table
Key attribute of entity set	Primary key of table
Derived attribute of entity set	*should not appear**

When translating ERs to schemas, we can use the following questions as a guide:

1. Can **PK** be used to uniquely identify other attributes
2. Is the LOWER BOUND satisfied in the schema?
3. Is the UPPER BOUND satisfied in the schema?

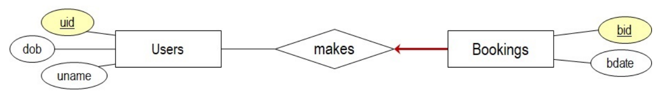
Many-to-Many




```sql
CREATE TABLE FlightInstances (
    fnum      INT REFERENCES Flights,
    fdate     DATE,
    aircraft  VARCHAR(10),
    PRIMARY KEY (fnum, fdate)
);
CREATE TABLE Bookings (
    bid   INT PRIMARY KEY
);
CREATE TABLE Includes (
    fdate DATE,
    fnum  INT,
    bid   INT REFERENCES Bookings,
    seat  VARCHAR(10),
    REFERENCES FlightInstances,
    PRIMARY KEY (fnum, fdate, bid)
);
```

1. Uniquely identify: `(fnum, fdate, bid)` able to ID the seat
2. Lower/Upper Bound: NA since it is a many-to-many

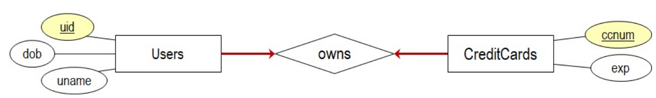
Many-to-One




```sql
CREATE TABLE Users (
    uid   INT PRIMARY KEY,
    uname VARCHAR(100),
    dob   DATE
);
-- Remove "Bookings", combine with "makes"
CREATE TABLE MakesBookings (
  uid   INT,
  bid   INT,
  bdate DATE,
  FOREIGN KEY (uid) REFERENCES Users (uid),
  -- no more REFERENCES Bookings (bid)
  PRIMARY KEY (bid)
);
```

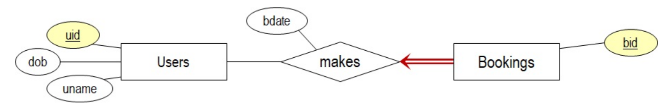
1. Uniquely identify: `(bid)` able to ID the `bdate`
2. Lower Bound: Able to have a booking without user. **PK** is `bid`, user has no constraints (can be `NULL`).
3. Upper Bound: Upper limit is 1. Unable to have booking belonging to > 1 user (`bid` is **PK**).

One-to-One




```sql
CREATE TABLE Users (
    uid INT PRIMARY KEY,
    ...
    REFERENCES CreditCards (ccnum)
); -- Assume CreditCards is not combined
CREATE TABLE CreditCards (
    ccnum INT PRIMARY KEY,
    ...
    REFERENCES Users (uid)
); -- Assume Users is not combined
```

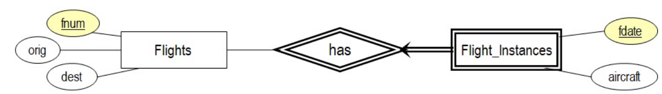
Key + Total Participation Constraint




```sql
CREATE TABLE Users (
    uid   INT PRIMARY KEY,
    uname VARCHAR(100),
    dob   DATE
);
CREATE TABLE MakesBookings (
  bid   INT PRIMARY KEY,
  bdate DATE,
  uid   INT NOT NULL,
  -- NOT NULL ensures that there must be a user!
  FOREIGN KEY (uid) REFERENCES Users (uid)
);
```

1. Uniquely identify: `(bid)` able to ID the `bdate`
2. Lower Bound: Unable to have a booking without user. `NOT NULL` ensures each `MakesBookings` entry always has a user.
3. Upper Bound: Upper limit is 1. Unable to have booking belonging to > 1 user (`bid` is **PK**).

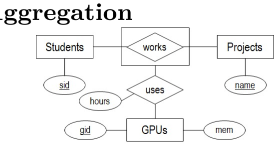
Weak Entity Sets




```sql
CREATE TABLE Users (
    fnum  INT PRIMARY KEY,
    orig  VARCHAR(10),
    dest  VARCHAR(10)
);
CREATE TABLE FlightInstances (
  fnum      INT REFERENCES Flights (fnum)
    ON DELETE CASCADE
    ON UPDATE CASCADE,
  fdate     DATE,
  aircraft VARCHAR(10),
  PRIMARY KEY (fnum, fdate)
);
```

Extended Notations - Aggregation

Abstraction that treats relationships as higher-level entities. Treat it as a relation class in OOP.

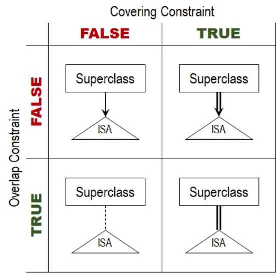


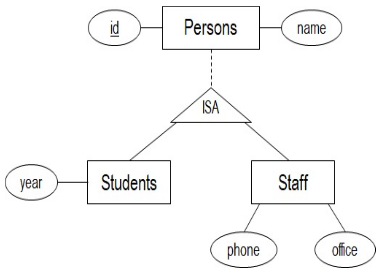

```sql
CREATE TABLE Uses (
    gid   INT REFERENCES GPUs,
    sid   INT,
    name  VARCHAR(50),
    hours NUMERIC,
    PRIMARY KEY (gid, sid, pname),
    FOREIGN KEY (sid, pname)
        REFERENCES Works (sid, pname)
);
```

Extended Notations - ISA Hierarchies

Every entity subclass is an entity in its superclass.

Overlap: Can a superclass belong to multiple subclasses?
Covering: Must a superclass belong to at least one subclass?






```sql
CREATE TABLE Persons (
    id     CHAR(20) PRIMARY KEY,
    name   VARCHAR(50)
);
CREATE TABLE Students (
    id     CHAR(20) PRIMARY KEY,
        REFERENCES Persons (id) ON DELETE CASCADE
    year   INT, -- or DATE?
);
CREATE TABLE Grads (
    id       CHAR(20) PRIMARY KEY
        REFERENCES Persons (id) ON DELETE CASCADE,
    phone    INT,
    office   VARCHAR(10)
);
```

SQL Queries

SELECT-FROM-WHERE

```sql
SELECT [ DISTINCT ] <target-list>
FROM <relation-list>
WHERE <conditions>
```

SQL Query → RA Expression → Query Execution Plan → Executable Code

EXAMPLE

```sql
SELECT DISTINCT a1, a2, ..., am
FROM r1, r2, ..., rn
WHERE c
```

RELATIONAL ALGEBRA:
$\pi[a1, a2, ..., am](\sigma[c](r1 \times r2 \times ... \times rn))$

Basic SQL Queries

- SELECT - Combine and process attribute values + Rename columns: Find restaurants, pizzas they sell and price of pizzas in SGD.

```sql
SELECT rname, pizza,'S$' || (price * 1.36) AS sgd
FROM Sells;
```

- WHERE - Condition for NULL values: `x IS NULL` vs `x <> NOT NULL`

IS NULL

```sql
SELECT cname
FROM   Customers
WHERE  area IS NULL;
```

cname
Adi
Yoga

2 rows

= NULL

```sql
SELECT cname
FROM   Customers
WHERE  area = NULL;
```

cname

0 rows

- WHERE - Pattern Matching
  - _ matches any single character
  - % matches any sequence of 0 or more characters
  - EXAMPLE:
    - Start with 'Ma', end with 'a':
      `WHERE pizza LIKE 'Ma%a'`
    - starts with 'A' and consists of at least 5 characters:
      `WHERE uname LIKE 'A____%'`
- SET OPERATIONS

```sql
Q1 UNION Q2
Q1 INTERSECT Q2 -- treats everything as distinct
Q1 EXCEPT Q2    -- treats everything as distinct

Q1 UNION  ALL Q2    -- do not remove duplicates
Q1 INTERSECT ALL Q2 -- do not remove duplicates
Q1 EXCEPT ALL Q2    -- do not remove duplicates
```

- JOIN OPERATIONS: Most common in practice: cross product + selection condition + attribute selection

```sql
SELECT DISTINCT rname, price
FROM Sells S JOIN Recipes R
    ON S.pizza = R.pizza
WHERE R.ingredient = 'Cheese';

SELECT DISTINCT rname, price
FROM Sells S NATURAL JOIN Recipes R
-- NATURAL JOIN Matches all common attributes
WHERE R.ingredient = 'Cheese';

SELECT DISTINCT C.cname
FROM Customers C LEFT JOIN Likes L
 ON C.cname = L.cname
WHERE L.pizza IS NULL; -- keep dangling tuples
```

Composing

Scalar Subqueries:

A query that returns at most a single value.

```sql
SELECT *
FROM Sells
WHERE price < (SELECT S.price FROM Sells S
               WHERE rname = 'Corleone Corner');
```

[NOT] IN:

Subquery **MUST** return exactly one column.

- `IN` returns `TRUE` if `<expr>` matches any subquery row
- `NOT IN` returns `TRUE` if `<expr>` matches no subquery row

```sql
WHERE <expr> [ NOT ] IN [ <subquery> | <tuple> ];
```

ANY/SOME

Subquery **MUST** return exactly one column. Expression `<expr>` is compared to each row from subquery using the operator `<op>`.

- `ANY` returns `TRUE` if comparison evaluates to `TRUE` for **at least one row** in the subquery

```sql
WHERE <expr> <op> ANY <subquery>;
```

ALL

Subquery **MUST** return exactly one column.Expression `<expr>` is compared to each row from subquery using the operator `<op>`.

- `ALL` returns `TRUE` if comparison evaluates to `TRUE` for **ALL** rows in the subquery

```sql
WHERE <expr> <op> ALL <subquery>;
```

[NOT] EXISTS

May return any number of columns

- `EXISTS` returns `TRUE` if the subquery returns at least one row
- `NOT EXISTS` returns `TRUE` if the subquery returns no row

```sql
WHERE [ NOT ] EXISTS <subquery>;
```

Scoping

- A table alias declared in a (sub-)query Q can only be used in Q or subqueries nested within Q
- If the same table alias is declared in both subquery Q_1 and in outer query Q_0 (or not at all), the declaration in Q_1 is applied

```sql
SELECT DISTINCT S1.rname
FROM Sells S1
WHERE price < ANY -- S1.price
    (SELECT price -- S2.price
    FROM Sells S2
    WHERE S2.rname = 'Lorenzo Tavern');
```

⋆ Not all constructs required: IN ≡ ANY ≡ EXIST

Ordering: ORDER BY

- `ORDER BY <attribute> ASC`: Ascending Order (default for SQL, ASC can be removed)
- `ORDER BY <attribute> DESC`: Descending Order
- If duplicate removal needed, attribute being sorted **must** appear in SELECT
- Sorting w.r.t multiple attributes / differing order supported

SQL

```sql
SELECT *
FROM   Recipes
WHERE  pizza IN
  (SELECT pizza
   FROM   Recipes
   WHERE  ingredient = 'Cheese')
ORDER BY pizza DESC, ①
         ingredient ASC; ②
```

within each pizza ang ASC

Result

pizza ① DESC	ingredient
Siciliana	Anchovies ② Asc
Siciliana	Caspers
Siciliana	Cheese
Margherita	Cheese ② Asc
Margherita	Tomato
Diavola	Cheese ② Asc
Diavola	Chilli
Diavola	Salami

Ordering: LIMIT and OFFSET

- `LIMIT k` Return the first* `k` rows of the result table
- `OFFSET i` Specify the position of the first row to be considered
- ⋆ `LIMIT` and `OFFSET` are typically meaningful only in combination with `ORDER BY`

Find the next 3 cheapest pizza, restaurant selling them and price.

```sql
SELECT *
FROM Sells
ORDER BY price ASC
OFFSET 3
LIMIT 3;
```

Aggregation

Function	Input Type	Output Type
MIN	*any comparable type*	*same as input*
MAX	*any comparable type*	*same as input*
SUM	Numeric data *(e.g., INT, BIGINT, REAL, etc)*	SUM(INT) → BIGINT; SUM(REAL) → REAL
COUNT	*any data*	BIGINT

Aggregate functions compute a ***single value*** from a set of tuples.

```sql
SELECT MIN(price) AS lowest
FROM Sells;
```

Query	Interpretation	Result
SELECT MIN(A) FROM R;	Minimum *non-NULL* value in *A*	0
SELECT MAX(A) FROM R;	Maximum *non-NULL* value in *A*	42
SELECT AVG(A) FROM R;	Average of *non-NULL* values in *A*	12 *(i.e., 48 / 4)*
SELECT SUM(A) FROM R;	Sum of *non-NULL* values in *A*	48
SELECT COUNT(A) FROM R;	Count of *non-NULL* values in *A*	4
SELECT COUNT(*) FROM R;	Count of rows in *A* (all rows)	5
SELECT AVG(DISTINCT A) FROM R;	Average of ***distinct*** *non-NULL* value in *A*	15
SELECT SUM(DISTINCT A) FROM R;	Sum of ***distinct*** *non-NULL* value in *A*	45
SELECT COUNT(DISTINCT A) FROM R;	Count of ***distinct*** *non-NULL* value in *A*	3

Left: R is an empty relation with attribute B
Right: S is a non-empty relation with n-rows with attribute A only having `NULL` values

Query	Result
SELECT MIN(B) FROM R;	NULL
SELECT MAX(B) FROM R;	NULL
SELECT AVG(B) FROM R;	NULL
SELECT SUM(B) FROM R;	NULL
SELECT COUNT(B) FROM R;	0
SELECT COUNT(*) FROM R;	0

Query	Result
SELECT MIN(A) FROM S;	NULL
SELECT MAX(A) FROM S;	NULL
SELECT AVG(A) FROM S;	NULL
SELECT SUM(A) FROM S;	NULL
SELECT COUNT(A) FROM S;	0
SELECT COUNT(*) FROM S;	*n*

Grouping

Application of aggregate functions are over all tuples of a relation

- Logical partition of relation into groups based on values for specific attributes
- Application of aggregation functions are now over each group (one result tuple for each group)

```sql
GROUP BY attr1, attr2, ...
```

⋆ f column A_i appears in SELECT, one condition must hold

- A_i appears in the GROUP BY clause
- A_i appears as input of aggregation function in SELECT clause
- The pkey of R appears in the GROUP BY clause

EXAMPLE

The first query is valid because it groups the results by the unique `rname` column from the `Restaurants` table and counts the number of `cname` values from the `Customers` table that have the same area value as the grouped `rname`. This is a valid aggregation because the `GROUP BY` clause includes all non-aggregated columns (i.e. `rname` and `area`) from the `Restaurants` table and the aggregated column (i.e. `cname`) from the `Customers` table.

The second query is not valid because it groups the results by the unique `rname` column from the `Restaurants` table and also includes the non-aggregated `cname` column from the `Customers` table in the `SELECT` clause. When using `GROUP BY`, all non-aggregated columns in the `SELECT` clause must also be included in the `GROUP BY` clause. In this case, the cname column from the `Customers` table is not included in the `GROUP BY` clause, so it is not clear which cname value should be displayed for each group of `rname` values.

```sql
SELECT R.rname, R.area,
       COUNT (C.cname)
FROM   Restaurants R, Customers C
WHERE  R.area = C.area
GROUP BY R.rname;
```

✓

```sql
SELECT R.rname, C.cname,
       COUNT (C.cname)
FROM   Restaurants R, Customers C
WHERE  R.area = C.area
GROUP BY R.rname;
```

✗

Having

`WHERE` clauses only works on each row. If we were to use `WHERE p < AVG(p)`, any p that does not fit this condition will be removed, resulting in `AVG(p)` to change. `HAVING` instead aggregates the condition for each group defined by `GROUP BY`.

```sql
GROUP BY attr1, attr2, ...
HAVING <condition>
```

⋆ If column A_i appears in HAVING, one condition must hold

- A_i appears in the GROUP BY clause
- A_i appears as input of aggregation function in HAVING clause
- The pkey of R appears in the GROUP BY clause

Conceptual Evaulation of Queries

FROM	Compute cross-product of all tables in FROM clause
WHERE	Filter (*i.e., keep*) tuples that evaluate to true on the WHERE condition(s)
GROUP BY	Partition table into groups w.r.t. the grouping attribute(s)
HAVING	Filter (*i.e., keep*) groups that evaluate to true on the HAVING condition(s)
SELECT	Remove all attribute(s) not specified in SELECT clause (*& remove duplicate if DISTINCT*)
ORDER BY	Sort tables based on specified attribute(s)
LIMIT/OFFSET	Filter (*i.e., keep*) tuples based on their order in the table

Conditional Expressions: CASE

Only one of the results will be returned, else is optional, and `NULL` returned if no condition matched.

```sql
CASE
    WHEN <condition_1> THEN <result_1>
    WHEN <condition_2> THEN <result_2>
    ELSE result_0
END
```

Conditional Expressions: COALESCE

Returns the first non-NULL value in the list of input arguments (order of `order` matters!)

```sql
COALESCE(<value_1>, <value_2>, <value_3>, ...)
// the above looks at <value_1> first, if it is
// non-NULL, it will return <value_1>
```

Conditional Expressions: NULLIF

Returns NULL if `<value_1>` = `<value_2>`, otherwise returns `<value_1>`

```sql
NULLIF(<value_1>, <value_2>)
```

```sql
SELECT
  NULLIF(1,1)
  AS val;
```

val
NULL

```sql
SELECT
  NULLIF(2,1)
  AS val;
```

val
2

```sql
SELECT
  NULLIF(NULL,1)
  AS val;
```

val
NULL

```sql
SELECT
  NULLIF(2,NULL)
  AS val;
```

val
2

Common Table Expression

- CTE_i is the name of a temporary table defined by query Q_i
- Each CTE_i can reference any other CTE_j that has been declared before CTE_i (i.e., $j < i$)
- The main SELECT statement Q_0 can reference any possible subset of all CTE_i → Any CTE_i not referenced can be deleted

```sql
WITH
CTE_1 AS ( Q_1 ),
CTE_2 AS ( Q_2 ),
...
CTE_n AS ( Q_n )
Q_0
-- main SELECT statement
```

Views

Used since we only usually need parts of the table, restrict access to table for certain users, and often use the same queries and subqueries frequently.

```sql
CREATE VIEW <name> AS
<query> -- SELECT statement
;
```

- A VIEW is a permanently named query, computation done each time the virtual table is accessed
- Results are not permanently stored, no actual table created.

Extended Concepts

Universal Quantification

QUERY: restaurant that sells all pizzas liked by 'Homer'

Transformation:

→ there does not exist pizza that 'Homer' Likes and not sold by the restaurant: all pizzas that 'Homer' likes and restaurant with rname = R does not sell

```sql
SELECT L.pizza
FROM Likes L
WHERE L.cname = 'Homer'
    AND NOT EXISTS (
        SELECT 1
        FROM Sells S
        WHERE S.pizza = L.pizza
        AND S.rname = 'R'
    );
```

→ Find no possible restaurants that does not sell any pizza liked by 'Homer'.

```sql
SELECT DISTINCT S1.rname
FROM Sells S1
WHERE NOT EXISTS (
    SELECT 1 FROM Likes L
    WHERE L.cname = 'Homer'
      AND NOT EXISTS (
          SELECT 1 FROM Sells S
          WHERE S.pizza = L.pizza
```

```
            AND S.rname = S1.rname
        )
    );
```

Cardinality

→ Find the set of pizzas sold by a restaurant that is a SUPERSET or EQUAL to that of the pizzas liked by 'Homer'.

```
SELECT DISTINCT rname
FROM Sells S
WHERE (
  SELECT COUNT(DISTINCT pizza)
  FROM (
    SELECT pizza FROM Sells S1 WHERE S1.rname = S.rname     // selecting correct R
    UNION
    SELECT pizza FROM Likes WHERE cname = 'Homer'     // pizzas Homer likes
    ) AS T1
  ) = (
    SELECT COUNT(DISTINCT pizza)
    FROM Sells S1 WHERE S1.rname = S.rname          // counting the number of pizzas
  );
```

Recursive Queries

```
WITH RECURSIVE
    CTE_name AS (
        Q_1
        UNION [ ALL ]
        Q_2 ( CTE_name )
    )
Q_0 ( CTE_name )
```

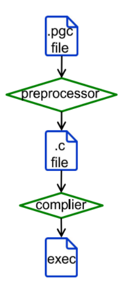
- Q_1 is non-recursive
- Q_2 is recursive and can reference CTE_{name}
- Query is evaluated 'lazily', stops when a fixxed-point is reached

Find all MRT stations can be reached from NS1 in at most 3 stops

```
WITH RECURSIVE
    Linker(to_stn, stops) AS (
        SELECT to_stn, 0
        FROM MRT
        WHERE fr_stn = 'NS1'
        UNION ALL
        SELECT M.to_stn, L.stops + 1
        FROM Linker L, MRT M
        WHERE L.to_stn = M.fr_stn
            AND L.stops < 2
    )
SELECT DISTINCT (to_stn)
FROM Linker;
```

Dynamic SQL

If we need to write different programs to access a database, we can mix a host language with SQL. The program is passed into a preprocessor, which returns a program in the host language. After which, this program is compiled into executable code.



Static SQL

```
int main() {
    EXEC SQL BEGIN DECLARE SECTION;      // 1.
      char supplName[30], prodName[20]; float price;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO @localhost USER john; // 2.

    // some code that assigns values to  // 3.
    // supplName, prodName, price;

    EXEC SQL INSERT INTO
      Sells(supplName, prodName, price)  // 4.
      VALUES(:supplName, :prodName, :price);
    EXEC SQL DISCONNECT; // 5.
    return 0;
}
```

1. Declares **shared** variables that can be used by both the host language and SQL statements
2. Specifies the database to connect to and ther username to use
3. Host part of the language
4. Inserts record into the table, must be preceded by a colon → this is reffered to as a static SQL
5. Closes the database connection

Dynamic SQL

The SQL statement to be executed is not fixed in advance. Instead, we store the statement in a string variable, which is then compiled and executed at runtime, checking the syntax at runtime.

- `EXEC SQL EXECUTE IMMEDIATE :stmt`, this part executes the SQL statement stored in `stmt`.

```
int main() {
    EXEC SQL BEGIN DECLARE SECTION;      // 1.
        const char *stmt = "INSERT INTO test VALUES(?, ?);";
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO @localhost USER john; // 2.

    EXEC SQL PREPARE mystmt FROM :stmt; // 3.

    EXEC SQL EXECUTE mystmt USING 42, 'foobar'; // 4.

    EXEC SQL DEALLOCATE PREPARE mystmt;  // 5.

    EXEC SQL DISCONNECT;
    return 0;
}
```

1. Init a SQL statement with 2 parameters
2. Specifies the database to connect to and ther username to use
3. Parsing the SQL statement and gives it a name `mystmt`
4. Executes SQL statement using 42 and 'foobar' as parameters
5. Releases the resources allocated to mystmt (EVERY `PREPARE` statement should be deallocated when no longer needed)

Call-Level Interface (CLI)

e.g. Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC). This is used when we do not want to mix the host langauges with SQL. Instead of using `EXEC SQL`, we instead call an API to run the queries.

ODBC Example in C++

```
void main() {
    char *sql;
    connection C("dbname= testbd user=postgre \
        password=cohondob hostaddr=127.0.0.1 port=5432"); // 1.

    // Some code here

    // Create SQL statement
    sql = "CREATE TABLE COMPANY (" \
      "ID INT PRIMARY KEY NOT NULL," \
      "NAME TEXT NOT NULL);";          // 2.

    work W(C); // Create a transactional object 3.

    W.exec( sql );    // 4.
    W.commit();
    C.disconnect();   // 5.
}
```

1. Creates a connection to the database
2. Assigns the SQL statement as a string to the `sql` variable
3. creates an object `W` for executing the SQL statement
4. Executes SQL statement recorded in `sql`
5. Closes database connection

Functions

Makes code reusable, easy to maintain, performance and acheives security (more in SQL injections). ISO standard: SQL/PSM (Persistant Stored Modules). PostgreSQL: PL/pgSQL

Example 1 —— Convert marks to letter grades

- $[70, 100] \rightarrow$ A
- $[60, 70] \rightarrow$ B
- $[50, 60] \rightarrow$ C
- $[0, 50] \rightarrow$ D

```
CREATE OR REPLACE FUNCTION convert(mark INT)
RETURNS char(1) AS $$
    SELECT CASE
        WHEN mark >= 70 THEN 'A'
        WHEN mark >= 60 THEN 'B'
        WHEN mark >= 50 THEN 'C'
        ELSE 'D'
    END;
$$ LANGUAGE sql;

/*
Query:SELECT convert (90);
Result: A

Query: SELECT Name, convert( Mark ) FROM Scores;

+-------+------+        +-------+---------+
| Name  | Mark |        | Name  | Convert |
+-------+------+        +-------+---------+
| Alice |   92 |        | Alice |    A    |
| Bob   |   63 | ====>  | Bob   |    B    |
| Cathy |   58 |        | Cathy |    C    |
| David |   47 |        | David |    D    |
+-------+------+        +-------+---------+
*/
```

Example 2 —— Return all tuples with highest mark

```
CREATE OR REPLACE FUNCTION topStudents()
RETURNS SETOF Scores AS $$
    SELECT *
    FROM Scores
    WHERE MARK = (SELECT MAX(Mark) FROM Scores);
$$ LANGUAGES sql;

/*
If we only want 1 student, we can instead do:
ORDER BY Mark DESC LIMIT 1 to find the TOP student.

Query: SELECT * FROM topStudents();

+-------+------+        +-------+------+
| Name  | Mark |        | Name  | Mark |
+-------+------+        +-------+------+
| Alice |   92 |        | Alice |   92  |
| Bob   |   63 | ====>  | David |   92  |
| Cathy |   58 |        +-------+------+
| David |   92 |
+-------+------+
*/
```

Example 3 —— Returning a new tuple

```
CREATE OR REPLACE FUNCTION topMarkCnt(OUT TopMark INT, OUT Cnt INT)
RETURNS RECORD AS $$
    SELECT Mark, COUNT(*)
    FROM Scores
    WHERE MARK = (SELECT MAX(Mark) FROM Scores)
    GROUP BY Mark;
$$ LANGUAGES sql;

/*
THis returns the highest mark and its number of occurrences

Query: SELECT topMarkCnt();

+-------+------+        +----------+-----+
| Name  | Mark |        | Top Mark | Cnt |
+-------+------+ ====>  +----------+-----+
| Alice |   92 |        |    92    |  2  |
| Bob   |   63 |        +----------+-----+
| Cathy |   58 |
| David |   92 |
+-------+------+
*/
```

Example 4 —— Return new set of tuples

```
CREATE OR REPLACE FUNCTION markCnt(OUT TopMark INT, OUT Cnt INT)
RETURNS SETOF RECORD Scores AS $$
    SELECT Mark, COUNT(*)
    FROM Scores
    GROUP BY Mark;
$$ LANGUAGES sql;

/*
Returns distinct Mark and number of occurrences

Query: SELECT markCnt();



+-------+------+        +----------+-----+
| Name  | Mark |        | Top Mark | Cnt |
+-------+------+ ====>  +----------+-----+
| Alice |   92 |        |    92    |  2  |
| Bob   |   63 |        |    63    |  1  |
| Cathy |   58 |        |    58    |  1  |
| David |   92 |        +----------+-----+
+-------+------+
*/
```

Difference between SETOF, RECORDS, TABLE

- `SETOF` is used when we want to return multiple records
- `RECORDS` is used for only 1 record, and it is not in the current schema
- `SETOF RECORDS` is used for multiple tuples, which are not in the current schema
- `TABLE` is similar to `SETOF RECORDS`, this means our earlier code in Example 4 can also be done with `TABLE`

```
CREATE OR REPLACE FUNCTION markCnt()
RETURNS TABLE(Mark INT, Cnt INT) AS $$
    SELECT Mark, COUNT(*)
    FROM Scores
    GROUP BY Mark;
$$ LANGUAGES sql;
```

Procedures

If no return value/tuple is needed we may use **SQL procedures** instead. To execute the following procedure, we use:
`CALL transfer('Alice', 'Bob', 100);`

Procedure Example

```
CREATE OR REPLACE PROCEDURE transfer(fromAcc TEXT, toAcct TEXT, amount INT)
AS $$
    UPDATE Accounts
    SET balance = balance - amount
    WHERE name = fromAcct;

    UPDATE Accounts
    SET balance = balance + amount
    WHERE name = toAcct;
$$ LANGUAGE SQL
```

SQL Functions/Procedures can be complex, having variables and *control structures*

- `IF ... THEN ... ELSE ... END IF`
- `LOOP ... END LOOP`
- `EXIT WHEN` – Used within a loop to exit the loop
- `WHILE ... LOOP ... END LOOP`
- `FOR ... IN ... LOOP ... END LOOP`

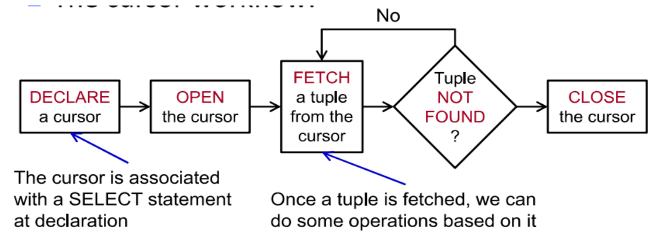
IF THEN Example

```
CREATE OR REPLACE FUNCTION swap(INOUT val1 INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
    temp_val INTEGER;
BEGIN
    IF val1 > val2 THEN
        temp_val := val1;
        val1 := val2;
        val2 := temp_val;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- `SELECT swap(99, 11);` → `(11, 99)`
- `SELECT swap(11, 99);` → `(11, 99)`

Cursors

Cursors are used to loop over a sorted sequence within the schema. If we need to loop over them and do some computation, we can use a cursor as a pointer to reference every variable in the table iteratively. A cursor enables us to access each individual row returned by a `SELECT` statement.




```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE(name TEXT, mark INT, gap INT) AS $$
DECLARE // 1.
    curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
    r RECORD;
    prv_mark INT;
BEGIN
    prv_mark := -1;
    OPEN curs;    // 2.
    LOOP
        FETCH curs INTO r;    // 3.
        EXIT WHEN NOT FOUND;  // 4.
        name := r.Name;       // 5.
        mark := r.Mark;
        IF prv_mark >= 0 THEN gap := prv_mark - mark;
        ELSE gap := NULL;
        END IF;
        RETURN NEXT;          // 6.
        prv_mark := r.Mark;   // 7.
    END LOOP;
    CLOSE curs;               // 8.
END;
$$ LANGUAGE plpgsql;
```

1. Declares cursor variable
2. Executes SQL statement, lets `curs` point to beginning of result
3. Read next tuple from `curs` and put into `r`
4. If `FETCH` operation did not get any tuple, terminate loop
5. If `FETCH` operation got tuple, compute values of name, mark and gap
6. Inserts a tuple `(name, mark, gap)` to the output of function
7. Updates `prv_mark` variable
8. Release resources allocated to `curs`

Other Cursor Movements

- `FETCH PRIOR FROM cur INTO r` → fetch the prior row
- `FETCH FIRST FROM cur INTO r` → fetch first row
- `FETCH LAST FROM cur INTO r` → fetch last row
- `FETCH ABSOLUTE 3 FROM cur INTO r` → fetch the 3^{rd} tuple

SQL Injections

Type of attack on dunamic SQL. Imagine `stmt` is the concatenation of 3 sub-strings:

`SELECT * FROM T WHERE Name ='` *input_name* `';`

Someone with malicious intent can instead put `a'; DROP TABLE T; SELECT 'a` into *input_name*. Now, `T` will be deleted once `stmt` is executed.

Triggers ↔ Trigger Functions

We want to implement this on an insertion, need the database to check this particular **condition** whenever appropriate.

- Expression condition about 'insertion occuring on Scores' → Trigger Functions
- Database to check this condition whenever appropriate → Triggers

```
// Trigger
CREATE TRIGGER scores_log_trigger
AFTER INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log_function();

// Trigger Function
CREATE OR REPLACE FUNCTION scores_log_function() ↩
    RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO Scores_Log(Name, EntryDate)
        VALUES (NEW.Name, CURRENT_DATE);
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

- Trigger tells database to watch out for insertions on Scores and call the `scores_log_func()` after each insertion of a tuple
- '`RETURNS TRIGGER`' indicates that this is a trigger function, only `RETURNS TRIGGER` allowed as only `TRIGGER` has access to the `NEW`.
- `NEW` refers to new row inserted into Scores
- `CURRENT_DATE` returns the current date

What else can a trigger function access?

- `TG_OP`: operation that activates trigger: `INSERT`, `UPDATE`, `DELETE`
- `TG_TABLE_NAME`: name of table that cause trigger invocation
- `OLD`: old tuple being updated/deleted

Trigger Timing Example

Whenever there is a insert/delete/update, insert tuple into `Scores_Log2` to record: name, operation performed, date of operation

```
// Trigger
CREATE TRIGGER scores_log_trigger2
AFTER INSERT OR DELETE OR UPDATE ON Scores
FOR EACH ROW EXECUTE FUNCTION scores_log2_func();

// Trigger Function
CREATE OR REPLACE FUNCTION scores_log2_function()
RETURNS TRIGGER AS $$ BEGIN
    IF (TG_OP = 'INSERT') THEN
        INSERT INTO Scores_log2
            SELECT NEW.Name, 'Insert', CURRENT_DATE;
        RETURN NEW;
    ELSEIF (TG_OP = 'DELETE') THEN
        INSERT INTO Scores_log2
            SELECT OLD.Name, 'Delete', CURRENT_DATE;
        RETURN OLD;
    ELSEIF (TG_OP = 'UPDATE') THEN
        INSERT INTO Scores_log2
            SELECT NEW.Name, 'Update', CURRENT_DATE;
        RETURN NEW;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

- `AFTER`: indicates that `score_log2_func()` is executed after the insertion on Scores is done
- `BEFORE`: `scores_log2_func()` would be executed **before** the insertion
- `INSTEAD OF`: `scores_log2_func()` would be executed **instead of** insertion

`BEFORE` Trigger Example

```
// Trigger
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW EXECUTE FUNCTION for_Elise_func();

// Trigger Function
CREATE OR REPLACE FUNCTION for_Elise_func()
RETURNS TRIGGER AS $$ BEGIN
    IF (NEW.Name = 'Elise') THEN
        NEW.Mark := 100
    ENDIF;
    RETURN ______;
END;
$$ LANGUAGE plpgsql;
```

1. `NEW`: Elise's mark would be 100 regardless of what we insert.
2. `NULL`: Signals to postgres that this execution has a problem, no tuple can be inserted. `RETURN NULL` in a `BEFORE` trigger tells database to ignore rest of operation.
3. `OLD`: No tuple can be inserted, for `INSERT`, `OLD` is initially set to `NULL`, thus `RETURN OLD` is same as `RETURN NULL`.
4. `OLD` execption: Whenever the function returns a **non-null** tuple, trigger would use it as a tuple to be inserted

```
CREATE OR REPLACE FUNCTION for_Elise_func()
RETURNS TRIGGER AS $$
BEGIN
    OLD.Name := 'Haha';
    OLD.Mark := 0;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Return Values of Trigger Function —— BEFORE

- `BEFORE INSERT`:
  - Returning a non-null tuple `t`, `t` will be inserted
  - Returning a null tuple, no tuple inserted

  `BEFORE UPDATE`:
  - Returning a non-null tuple `t`, `t` will be updated tuple
  - Returning a null tuple, no tuple updated

  `BEFORE DELETE`:
  - Returning a non-null tuple `t`, deletion proceeds as normal
  - Returning a null tuple, no tuple deleted

Return Values of Trigger Function —— AFTER

- `AFTER INSERT`, `AFTER UPDATE`, `AFTER DELETE`, does not matter what is returned, as trigger function is invoked **after** main operation is done.

`INSTEAD OF` Trigger Example

This type of triggers can be defined as `VIEW`s only. Instead of doing something on a view, do it on a table. e.g. Whenever someone wants to update tuple in `Max_Score` (`VIEW`), update corresponding tuple in `Scores`.

```
// Trigger
CREATE TRIGGER update_max_trigger
INSTEAD OF UPDATE ON Max_Score
FOR EACH ROW EXECUTE FUNCTION update_max_func();

// Trigger Function
CREATE OR REPLACE FUNCTION update_max_func()
RETURNS TRIGGER AS $$
BEGIN
    UDPATE Scores
    SET Mark = NEW.Mark WHERE Name = OLD.Name;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

- Returning `NULL` signals the database to ignore the rest of the operation on the current row.
- Returning a non-null tuple signals the database to proceed as normal.

Trigger Levels

`FOR EACH ROW`: **row-level** trigger that executes trigger function for every tuple encountered.

`FOR EACH STATEMENT`: **statement-level** trigger that executes the trigger function only once

```
// Trigger
CREATE TRIGGER del_warn_trigger
BEFORE DELETE ON Scores_Log
FOR EACH STATEMENT EXECUTE FUNCTION del_warn_func();

// Trigger Function
CREATE OR REPLACE FUNCTION del_warn_trigger()
RETURNS TRIGGER AS $$
BEGIN
    RAISE NOTICE 'Not supposed to delete from log.';
    RETURN NULL; /* db prompts on deletion attempt */
END;
$$ LANGUAGE plpgsql;
```

- Statement-level triggers ignores the values returned by trigger functions.
- `RETURN NULL` would not make database omit subsequent operations

`EXECPTION` Example

```
// Same Trigger as above
// Trigger Function
CREATE OR REPLACE FUNCTION del_warn_trigger()
RETURNS TRIGGER AS $$
BEGIN
    RAISE EXCEPTION 'No deletion from log allowed.';
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

- `INSTEAD OF` only allowed on **row-level**
- `BEFORE/AFTER` allowed on both **row** and **statement-level**

Trigger Condition

```
// Trigger
CREATE TRIGGER for_Elise_trigger
BEFORE INSERT ON Scores
FOR EACH ROW WHEN (NEW.Name = 'Elise')
EXECUTE FUNCTION for_Elise_func();

// Trigger Function
CREATE OR REPLACE FUNCTION for_Elise_func()
RETURNS TRIGGER AS $$
BEGIN
    NEW.Mark := 100;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

In general, the condition in `WHEN()` could be more complicated, 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`

Deffered Trigger

Scenarios where we need to **defer** checking of triggers. e.g. trigger on `INSERT/UPDATE/DELETE` checking total balance of each customer Accounts. Total balance should be at least **150**.

- Alice transfer 100 from Account 1 to Account 2, assuming both have a starting balance of 100. 2 update statements, deduct 100 from Account 1, add 100 to Account 2
- Trigger requirement violated in first update statement
- ***Solution:*** Put the 2 update statements in 1 transaction, defer the trigger check to the end of the transaction

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFFERABLE INITIALLY DEFFERED
FOR EACH ROW EXECUTE FUNCTION bal_check_func();
```

- `CONSTRAINT` and `DEFERRABLE` together indicate that the trigger can be deffered
- `INITIALLY DEFFERED` indicates that by default, trigger is deffered. `INITIALLY IMMEDIATE` trigger not deffered by default.

DEFFERED Triggers only work for `AFTER` and `FOR EACH ROW`

- `AFTER` to defer the trigger, has to be allowed to execute after main operation

With the above deffered trigger, we can do the following, as the trigger will only be activated at `COMMIT`

```
BEGIN TRANSACTION;
UPDATE ACCOUNT SET Bal = Bal - 100 WHERE AID = 1;
UPDATE ACCOUNT SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

INITIALLY IMMEDIATE Example

```
... DEFFERABLE INITIALLY IMMEDIATE ... /*Trigger*/
BEGIN TRANSACTION;
SET CONSTRAINTS bal_check_trigger DEFFERED
UPDATE ACCOUNT SET Bal = Bal - 100 WHERE AID = 1;
UPDATE ACCOUNT SET Bal = Bal + 100 WHERE AID = 2;
COMMIT;
```

- Multiple triggers defined for same event on the same table
- Order of trigger activation
  - `BEFORE` statement-level triggers
  - `BEFORE` row-level triggers
  - `AFTER` row-level triggers
  - `AFTER` statement-level triggers
- Within each category, triggers activated in alphabetic order
- If a `BEFORE` row-level trigger returns `NULL`, then subsequent triggers on same row are **omitted**.

Functional Dependencies

A *normal form* is a definition of minimum requirements to reduce data redundancy and improve data integrity.

- An example of a FD can be: denoted as NRIC → Name.
- NRIC decides Name, if 2 records have the same NRIC, then they will always have the same name.
- To identify this: find a contradicting example: e.g. No 2 shops sell the same product:
- Contradicting example: Shop1 and Shop2 both sells ProductA, not allowed $\therefore P \rightarrow S$

Armstrong's Axioms

1. **Reflexivity:** AB → A
2. **Augmentation:** If A → B, then AC → BC
3. **Transitivity:** If A → B, and B → C then A → C
4. **Decomposition:** If A → BC then A→ B and A → C
5. **Union:** If A → B and A → C then A → BC

Closure

Four attributes *A, B, C, D*. Given that B → D, DB → A, AD → C, check if B → C holds.

- $\{ B \}^+ = \{ B \}$
- $\{ B \}^+ = \{ B, D \}$, since B → D
- $\{ B \}^+ = \{ B, D, A \}$, since DB → A
- $\{ B \}^+ = \{ B, D, A, C \}$, since AD → C
- { B, D, A, C } reffered to as *closure* of B, which is the set of componenets that can be 'activated' by B

Similarly to prove that something, e.g. X → Y does not hold, we just need to show $\{ X \}^+$ **does not contain** Y.

Superkeys / Keys of a table

Superkeys: set of attributes that decides all other attributes
Keys: Superkey that is minimal. Whether or not a table has *redundancy* or *anomalies* depend on keys.

Finding Keys

- Given a list of FDs, any attribute that *does not appear* on the RHS, it means that attribute **must be in every key**!
- Given a key: e.g. { A, C } is able to activate all other attributes, we do no need to check its supersets for *keys* since keys have to be minimal! e.g. no need to check { A, C, B }, { A, C, D }, { A, C, B, D } etc.

Normal Forms

BCNF requires that if there is any **non-trivial** and **decomposed** FD $A_1A_2...A_n \rightarrow B$, then $A_1A_2...A_n$ must be a superkey. More simply, BCNF requires LHS to be super keys, prevent redundancy.
Decomposed FD: FD whose RHS only has 1 attribute
Non-Trival & Decomposed FD: Decomposed FD whose RHS does not appear in LHS

1. Compute closure of each attribute subset
2. Derive keys of R (using closures)
3. Derive non-trivial and decomposed FDs from each closure
4. For each of the FD, if all LHS is super key then R satisfy BCNF

Simplified BCNF

We have violation of BCNF iff we have a closure that satisfies the **more but not all** condition.

- Compute closure of each attribute subset
- If there is a closure (as below), then R is NOT in BCNF:
  - Closure contains some attributes not in $\{ A_1A_2...A_n \}$
  - Closure does not contain all attributes in table (more but not all closure)

Normalization Algorithm

1. Find subset X of attributes in R, suhc that its closure $\{X\}^+$ (i) does not contain more attributes than X, but (ii) does not contain all attributes in R
2. Decompose R to form 2 tables R_1 and R_2 such that
   - R_1 contains all attributes in $\{X\}^+$
   - R_2 contains all attributes in X and attributes not in $\{X\}^+$
3. If R_1 nto in BCNF¡ further decompose R_1, same for R_2

BCNF decomposition not unique. If table only has 2 attributes, it MUST BE in BCNF: Check ∅ set, A→B, B→A or both.

Projection of Closures/FDs

If we derive closures on table R_i decomposed from table R, we can enumerate attribute subsets of R. For each subset, derive its closure on R, thereafter project each closure onto R by removing those attributes that do not appear in R_i.

Properties of BCNFs

No `UPDATE/DELETE/INSERT` anomalies. Small redundancy. Original table can be reconstructed from decomposed table.

- BCNF gurantees lossless join whenever common attributes in R_1 and R_2 contain a superkey of R_1 or R_2
- Decomposing R, R_1 contains all attributes in $\{X\}^+$, R_2 contains all attributes in X as well as those not in $\{X\}^+$
- Thus X will be the common attributes between both R_1 and R_2, X will be a superkey of R_1

However, dependencies may not be preserved.

3NF

A table satisfies 3NF if and only if for every non-trivial and decomposed FD:

- Either the LHS is **superkey**
- OR RHS is a **prime attribute** (appears in a key)
- 3NF more lenient than BCNF:
  - Satisfy BCNF → satisfy 3NF (not necessarily vice versa)
  - Violate 3NF → violate BCNF (not necessarily vice versa)

FD Equivalence: Prove that Given *S*, *S'* can be derived, and vice versa.

3NF Check

1. Derive keys of table R
2. For each FD, check against LHS superkey OR each attribute on RHS is prime
3. If all FDs satisfy, then R is in 3NF

3NF Decomposition Algorithm

1. Given table R and set S of FDs, Derive minimal basis of S
2. In minimal basis, combine FDs whose LHS are the same
3. Create a table for each FD remained
4. If non of the tables contains a key of original table R, create table that contains a key of R (any key of R is ok)
5. Remove redundant tables

Minimal Basis

Given set S of FDs, minimal basis of S, M is a simplified version of S, such that M equivalent to original set S.

1. every FD in S can be derived from M, and vice versa
2. every FD in M is a non-trivial and decomposed FD
3. if any FD is removed from M, then some FD in S cannot be derived from M
4. for any FD in M, if we remove an attribute from its left hand side, then the FD cannot be derived from S

Adding Key for Lossless Join

R(A, B, C, D), with A→B, C→D

- Minimal basis: A→B, C→D
- Corresponding tables: R_1(A, B), R_2(C, D)
- Notice that R_1 and R_2 cannot be used to reconstruct R
- This is why we require the following:
  - Check if any of the tables contain a key for R; if not, then create a table that contains a key for R
- In this case, R has only one key: AC ☆
- Therefore, we add a table R_3(A, C)

Use the key of original table to "guide" the JOIN back