- 'NULL IS NULL' = TRUE, '1 IS NOT NULL' = TRUE
- '1 IS NULL' = FALSE, 'NULL IS NOT NULL" = FALSE
- 'COALESE(NULL, 1, 2, NULL, NULL)' returns 1
- 'COALESE(NULL, NULL, NULL)' returns NULL

CS2102 AY24/25 Sem2 Finals Cheatsheet

Order of SQL Query Execution: **1. FROM > 2. JOIN > 3. WHERE > 4. GROUP BY > 5. HAVING > 6. SELECT > 7. ORDER BY**

Common SQL Keywords:

Keyword	Use Case	Keyword	Explanation
BEGIN	Scope Transaction	**>, <, >=, <=**	Basic arithmetic comparison
END	Scope Transaction	**=, !=,<>**	Basic arithmetic equality checks. != same as <>
COMMIT	Scope Transaction	**IN**	Used to check if a value matches any values in a list. X IN ("aa", "bb") retrieves row where X is aa or bb.
ABORT	Scope Transaction		
ROLLBACK	Scope Transaction	**||**	• || is a concatenation operator
NOT NULL	Integrity Constraints	**LIKE, ILIKE**	Pattern matching. LIKE → Case sensitive. ILIKE → Case insensitive. (See pattern matching cheatsheet)
PRIMARY KEY	Integrity Constraints	**BETWEEN … AND**	• Used to filter values within a range. • E.g date BETWEEN '2024-01-01' AND '2024-12-31'. Date ranges are inclusive. [ ]
UNIQUE	Integrity Constraints		
FOREIGN KEY	Integrity Constraints	**DROP**	Modification
CHECK	Integrity Constraints	**ON UPDATE**	Referential Action
		ON DELETE	Referential Action
DELETE	Modification	**COALESE (…)**	Returns the first non-NULL item (from left to right)

Pattern	Description	Example
%	Matches zero or more characters	WHERE name LIKE 'A%' matches any value starting with "A"
%%	Matches any string (including empty)	WHERE description LIKE '%%' matches all descriptions.
%word%	Matches strings containing "word"	WHERE comment LIKE '%error%' matches "Runtime error".
word%	Matches strings starting with "word"	WHERE tag LIKE 'Admin%' matches "Administrator", "Admin".
%word	Matches strings ending with "word"	WHERE tag LIKE '%Admin' matches "SuperAdmin", "Admin".
_	Matches exactly one character	WHERE name LIKE 'J_n' matches "Jan", "Jon", "Jim", etc., XX John
___	Matches exactly 3 characters	WHERE code LIKE '___' matches "ABC", "123", but not "ABCD".
[ ]	Matches any single char within []	WHERE name LIKE 'B[aeiou]g' matches "Bag", "Beg", "Big", etc.
[^] or [!]	Matches any single character not in []	WHERE name LIKE 'B[^aeiou]g' matches "Bag", "Beg", but not vowels in the middle.
-	Matches a range of characters	WHERE name LIKE '[A-C]%' matches values starting with "A", "B", or "C" (e.g., "Anna", "Charlie").
\	Escape character for special symbols	WHERE name LIKE '100\%' ESCAPE '\' matches values like "100%" literally instead of treating % as a wildcard.

Simple Query Operations

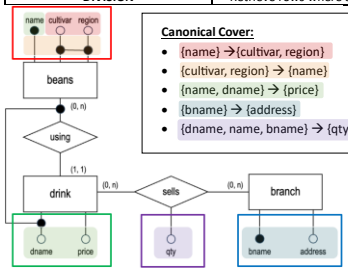
Keyword	Use Case
SELECT-FROM-WHERE →	Data Retrieval
*	Select ALL
ORDER BY … ASC/DESC →	Sort Data
DISTINCT	Remove Duplicates
AS	Rename/Alias
CASE-WHEN-THEN-ELSE	• CASE: Start of Conditional Expression • WHEN <condition a> THEN <branch a> • ELSE <optional branch • END: End of CASE expression
COUNT	Count rows
CROSS JOIN	Cartesian product **CROSS JOIN** is useful in scenarios where you need **every possible combination** of two sets of data. Generate Test Data/Combinations Creating Date Ranges / Time Slots

Aggregate Query Operations

Keyword	Use Case
COUNT()	Count NUM of non-null rows
SUM()	Calculate the sum of a Col
MAX()	Get the maximum value
MIN()	Get the minimum value
AVG()	Calculate the average value
STDDEV()	Calculate the STD DEV
TRUNC()	Truncate a number or date to a specific precision
GROUP_CONCAT()	Concatenate values from a group of rows
PERCENTILE_CONT()	Calculate percentile
HAVING	Filter groups after aggregation
DISTINCT	Remove duplicate values
GROUP BY	Group rows by a specific cols
VARIANCE()	Calculate the variance of a set of values
IS NULL / IS NOT NULL	NULL checks

Algebraic Query Operations:

Keyword	Use Case
CROSS JOIN	Combine every row from one table with every row from another
INNER JOIN/ JOIN	Combine rows from two tables where a condition is met
NATURAL JOIN	Join tables on columns with the same name and type
OUTER	Generic term for joins that return unmatched rows
LEFT OUTER JOIN/ LEFT JOIN	Return all rows from the left table and matched rows from the right table
RIGHT OUTER JOIN / RIGHT JOIN	Return all rows from the right table and matched rows from the left table
FULL OUTER JOIN / FULL JOIN	Return rows when there is a match in either table
UNION*	Combine results from two queries, eliminating duplicates
INTERSECT*	Return rows common to both queries
EXCEPT*	Return rows from 1st query that do not exist in the 2nd
DIVISION	Retrieve rows where a value in one table appears for all values in another table

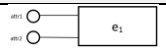


Canonical Cover:
- {name} →{cultivar, region}
- {cultivar, region} → {name}
- {name, dname} → {price}
- {bname} → {address}
- {dname, name, bname} → {qty}

FDS:
- {name} →{cultivar, region}
- {cultivar, region} → {name}
- {bname} → {address}
- {dname, name} → {price}
- {dname, cultivar, region} → {price}
- {dname, name, bname} → {qty}
- {dname, cultivar, region, bname} → {qty}

Entity-Relationship (ER) Diagram

Attributes

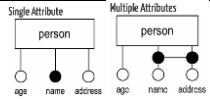


Attributes, Values, & Value Sets
- The ER model is value-oriented
- Values can be Integers, Strings, or Atoms.
- Attributes → The circles connected to rectangles.
  - Attributes are mainly used to help distinguish between the entities.

Attributes of Entities

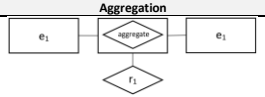
Entities can have attributes. All entities in an entity set have same attributes, but take diff values.

Primary Keys (Single, Multiple)



Entities' Identity
- One or more attributes can identify the entity.
- This is a property of all entities in an entity set.
  - At the very least, the combination of all attributes identifies the entity uniquely.
- Use the black circle/underline name to differentiate key attributes
  - The set of ALL identities → candidate keys.
  - Only entity sets can have identities in ER diagram (i.e no black dots on a r/s set)

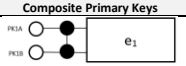
Aggregation



Aggregation: Associating with Relationship Set
- In some instance, we want to associate an entity set with a relationship set. Represented by wrapping the relationship set in a box.

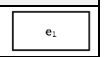
Simple Rule:
- A box can only be connected to a diamond; A diamond can only be connected to a box.
- If you want to connect 2 diamonds → Need to aggregate → Wrap with box

Composite Primary Keys

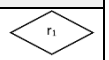


- Candidate Keys (CK)→A set of attributes,UNIQUE NN
- Ideally, use the **minimal** set of attributes. (NO need to be minimum)
- Minimal → Cannot remove anything anymore, where it can still uniquely identity an entity.
- E.g $CKs = \{\{a, b\}, \{c\}\}$
- Both $\{a, b\}$ & $\{c\}$ are minimal, but $\{c\}$ is the min.

Entity/Entity Set
- Entities are identifiable things.
- The named box → a set of entities/entity set.



Relationship Set
- Relationship associates 2 entities (fewer/more).
- The named diamond represents a set of relationships or a relation set.



Association
- A relationship set is a set of relationships associating entities from the same entity sets.
- In the example (person, contract, company). We exclude attributes for simplicity.
  - As it is a set, a person P can only be associated with a company C at most once.
  - But they can be associated with different company/person.

Distinguishing Relationships
- Relationship are distinguished not by their attributes, but by their participating entities.
- The "pair" or n-ary MUST be UNIQUE

Attributes of Relationships:
- Relationship can have attributes.
- All relationships in one relationship set have the same attributes (but different values)
- The attributes are dependent on the entities being associated.

Associating the Same Entity Set:
- Relationships can associate entities from the same entity set. In general, participation/role in the relationship can be named.
  - There may be a recursive/unary relationship.
  - The relation name is used as an additional node to make the relationships clearer btwn entities

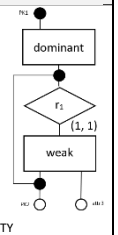
N-Ary: More than 2 Entity Sets
- Relationship sets can associate more than 2 entity sets. (n-ary relationship/ relationship sets)

Weak Entity Set

Partial Identification (Partial Keys)
- Some entries can only be identified within the scope of a relationship with another entity set.
- The relationship MUST exist and be UNIQUE for each entity in the set.

Weak Entities
- Matric numbers are given by the universities.
- The same number can be used by different uni.
  - University is a DOMINANT ENTITY
  - Student is a WEAK ENTITY.

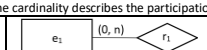
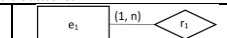
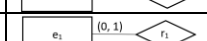
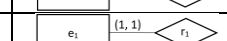
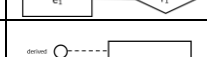
Student is a partial key here, cannot be identified by its attributes alone.



Cardinalities

Kinds of Participation
- The cardinality of the participation in a relationship can be constrained by a min & max value as (min, max).
- If an arbitrary letter is used, e.g n, m. It means that the min/max is unbounded. → Usually used for max value.

Classifications: The cardinality describes the participation of the entity set it is attached.

Unconstraint		**Total Participation**	
Key Constraint		**Key + Total Participation**	
Derived Attributes		• (1,x) → Mandatory Participation • (0,x) → Optional Participation • (x,1) → 1-to-1 r/s → If (x,1) for all entities involved → 1-to-many → If (x,1) for 1, else (x,n) • (x,n) → Many-to-Many r/s → if (x,n) for all	

3 Rules
- **Rule 1: Attributes**
  - Value set are mapped to domains
  - Attributes are mapped to Entities
- **Rule 2: Entity Set (PK)**
  - Entity Sets are mapped to relations
  - Entity Set attributes are mapped to PK and UNIQUE NOT NULL.
- **Rule 3: Relationship Sets (Composite PK, FKs)**
  - Relationship sets mapped to relations

3 Exceptions
- **Exception 1: One-To-Many (Remove from PK)**
  - Relax 1 of the PK to use NOT NULL
  - PK (A,B) → PK(A), B NOT NULL
- **Exception 2: One-To-One (Merge Tables)**
  - Combine tables A & B, if you get a chicken-&-egg situation → Resolves minimality constraints
- **Exception 3: Weak Entity (Add to PK)**
  - Partial Keys are not Unique, should add the primary key of dominant set entity:
  - PRIMARY KEY(dominant, partial)

- Attributes are the columns in the table
- Box = Table
- Black Circle = Primary Key / UNIQUE NOT NULL
- Many black circles = Candidate Key
- Diamond → Association, need to form pairs.

Nested Query Operations:

Keyword	Use Case	Explanation
CREATE TABLE	Splitting Query, Permanent Copy	We can make a copy of a subquery in a new table.
CREATE TEMPORARY TABLE	Splitting Query, Temporary Copy	We can make a copy of a subquery in a temporary table. Note that this temporary table only exist for the duration of the data base session. (Exist until you quit pgAdmin)
CREATE VIEW	Splitting Query, View	**Example:** CREATE VIEW active_users AS SELECT * FROM users WHERE status = 'Active'; (Creates a view of active users). **Note:** Views are unmaterialized, meaning they don't store data. A **Materialized View** stores results & needs manual refresh.
WITH … AS (…)	Splitting Query, Single Query, CTE WITH name1 AS (query1), name2 AS (query2), … namek AS (queryk)	**Common Table Expression (CTE)** is a copy of a subquery in a temporary table that **only exist for the query**. CTE is a VIEW that only exist for 1 Query. **Example:** WITH active_users AS (SELECT * FROM users WHERE status = 'Active') SELECT * FROM active_users; (Defines and queries a temporary result set).
FROM (subquery) AS …	Splitting Query, Single Query, FROM	Subqueries can also be done via the FROM operator. **Example:** SELECT avg_salary FROM (SELECT department_id, AVG(salary) AS avg_salary FROM employees GROUP BY department_id) AS dept_avg WHERE department_id = 1;
SELECT (Scalar subquery)	Splitting Query, Scalar Subquery, SELECT	• We can use subquery in the SELECT clause. • HOWEVER, it MUST return ONLY 1 column & 1 row • If there are no values in query → Return NULL • If there are > 1 values in query → Return runtime error **Example:** SELECT name, (SELECT COUNT(*) FROM orders WHERE orders.customer_id = customers.id) AS total_orders FROM customers; (Returns each customer's name along and order count).
WHERE … KEYWORD (subquery)	Nesting Query, Computing Tuples, **KEYWORD: IN**	We can use subquery with the WHERE clause to compute the tuples for IN clause. Can be used similarly in Nested Query.
	Nesting Query, Equals to Any, **KEYWORD: = ANY**	ANY is any that matches the condition. Never use comparison to a subquery without specifying ALL or ANY quantifiers!!!
	Nesting Query, Outer Join, Except, Aggregate **KEYWORD: = ALL**	ALL adds expressive power, similar to OUTER JOIN, EXCEPT and aggregate functions.
WHERE EXISTS (query / subquery) OR NOT EXISTS	Nesting Query, Empty/Not Empty, SELECT TABLE FROM ATTRIBUTE a WHERE EXISTS (COND);	• The EXIST clause evaluates to True if non empty, else false Note: (Typical scoping rules still apply!) • If the subquery is correlated to the query, the subquery is known as a correlated subquery. • For correlated subquery, it will be computed multiple times • E.g COND: SELECT … FROM … WHERE a.a = b.a …
WHERE cond (query)	Nesting Query, Correlation, Subquery, Correlated Subquery	• All subqueries can be correlated. • It is slightly more powerful than aggregate functions, as it enables us to see all prices of games with different versions but same name. Example: -- Nested Query: WHERE g1.price >= ALL (SELECT g2.price FROM…WHERE…) -- Aggregate Query: WHERE g1.price = (SELECT MAX(g2.price) FROM … WHERE …) • The **aggregate function (MAX) is typically more efficient** since it avoids multiple comparisons. • The **nested query using ALL allows more flexible conditions**, e.g., price > ALL(…) to find games that **exceed** all other versions rather than just matching the max.
SELECT… FROM… WHERE name IN (query)	Nesting Query, Correlation, Nested Scoping ** This is similar to lexical scoping. • We can use columns from an outer table in an inner query, but not another way round. SELECT (SELECT c.last_name FROM customers c WHERE c.country = 'Singapore' AND d.customerid = c.customerid), d.name FROM downloads d;	
Negated Queries (Nested + Negation)	Nested Queries are powerful when combined with negation	
SELECT … FROM… GROUP BY … HAVING …	Nesting Query, Nested Having (SQL below finds the countries with the largest #customers) • Nested queries may be necessary if we are using aggregation functions on 2 different groupings SELECT c1.country FROM customers c1 GROUP BY c1.country HAVING COUNT(*) >= ALL (SELECT COUNT(*) FROM customers c2 GROUP BY c1.country);	
LIMIT	Restrict the number of rows returned in a query result. SELECT * FROM books LIMIT 10; returns only the first 10 records.	
OFFSET	Skip a specific number of rows before starting to return results. SELECT * FROM books LIMIT 10 OFFSET 20; skips the first 20 rows and retrieves the next 10.	

Relational Algebra

Select Operator (σ)
- $\sigma_{[c]}(R)$ **selects** all rows from the relation R that satisfies the selection condition **c.**
- This is similar to **WHERE** clause in SQL.
- Only the row whose condition which evaluates to true will be kept.

Renaming Operator (ρ)
- $\rho(R_1, R_2)$ can be used to **rename** the relation.
  - This do not create a new relation in the hard disk, but we can simply refer to this table as R_2.
- $\rho(R_1, R_2(A_1 \to A_2))$: used to rename the relation & some of its attributes.
  - $(A_i \to B_i$ renames attribute A_i into B_i similar to **AS** keyword.)
- Can use dot notation to simplify writing as well:

Projection Operator (π)
- $\pi_{[l]}(R)$ **keeps** only the columns specified in the ordered list **l** and in the same order.
- This is similar to **SELECT** clause in SQL.
  - Note: Duplicates are automatically dropped in **π**
- In relational algebra, we CANT add new columns, unlike queries
- Note: QUERY uses hash aggregate to remove duplicates with DISTINCT (set).

Join(⋈)
- $R_1 \bowtie_{[c]} R_2$ is simply defined as $\sigma_{[c]}(R_1 \times R_2)$.
- In other words, we include only tuples that **satisfies the condition c** after the cross product.

Join Variants

	NEW	OLD
INSERT	✅	❌
UPDATE	✅	✅
DELETE	❌	✅

- If OLD is returned on insert → Since INSERT doesn't have OLD values, NULL will be returned instead. Similarly for NEW in DELETE
- If RETURN NULL is in BEFORE TRIGGER → Skip the triggering operation ONLY.

Relational Algebra	SQL Equivalent
π[r.attr](σ[c](ρ(rel, r)))	SELECT DISTINCT r.attr FROM rel r WHERE c;

Cross Product(x)

$R_1 \times R_2$ combine each row of R_1 with each row of R_2 and keep the n columns of R_1 and the m columns of R_2.

Natural Join	• If we exclude the condition **c**, the operator becomes the **natural join operator** (i.e., ⋈). • For example, R1 ⋈ R2
Outer Join	• We also have **left** ($\ltimes_{[c]}$), **right** ($\rtimes_{[c]}$), and **full** ($\overset{\ltimes}{\rtimes}_{[c]}$) outer join variants depending on COND **c**. • There is also natural variant.

Set Operators

Operation	Relational Algebra	SQL Equivalent
$R \cup S$	$\pi_{[c]}(R) \cup \pi_{[c]}(S)$	SELECT * FROM R **UNION** SELECT * FROM S
$R \cap S$	$\pi_{[c]}(R) \cap \pi_{[c]}(S)$	SELECT * FROM R **INTERSECT** SELECT * FROM S
$R - s$	$\pi_{[c]}(R) - \pi_{[c]}(S)$	SELECT * FROM R **EXCEPT** SELECT * FROM S

- The 2 relations MUST BE union-compatible E.g Same column types, Same number of columns etc.
- We also use the walrus operator (≔), similar to an assignment to simplify complex queries.
  - The temporary relation can be used for subsequent algebraic operations.

Example: ** UNION, INTERSECT, and EXCEPT automatically removes duplicates → NO NEED DISTINCT

Find all the different pairs of customer name and restaurant name such that they are in the same area.	
π[c.cname, r.rname](σ[c.area = r.area](ρ(customer, c) × ρ(restaurant, r)))	SELECT c.cname, r.rname FROM customer c, restaurant r WHERE c.area = r.area;
Find all the different restaurant name (rname), pizza, and the price of the pizza sold by restaurants in London	
π[r.rname, s.pizza, s.price](σ[r.rname = s.rname ∧ r.area = 'London'](ρ(restaurant, r) × ρ(sells, s)))	SELECT r.rname, s.pizza, s.price FROM restaurant r, sells s WHERE r.rname = s.rname; AND c.area = 'London';
Find the different pizza sold by Bella Italia but not Desert Diner	
Q1 := π[pizza](σ[rname = 'Bella Italia'](sells)) Q2 := π[pizza](σ[rname = 'Desert Diner'](sells)) Q1 ∩ / ∪ / − Q2	SELECT s.pizza FROM sells s WHERE s.rname = 'Bella Italia' INTERSECT / UNION / EXCEPT SELECT s.pizza FROM sells s WHERE s.rname = 'Desert Diner';
Find all the different pairs of customer name and restaurant name such that they are in the same area.	
π[c.cname,r.rname](ρ(customer, c) ⋈[c.area = r.area] ρ(restaurant, r))	π[c.cname, r.rname](-- Cross Product Form σ[c.area = r.area](ρ(customer, c) x ρ(restaurant, r))) π[c.cname, r.rname](-- Natural Join Form customer ⋈ restaurants)	SELECT c.cname, r.rname FROM customer AS c JOIN restaurant AS r ON c.area = r.area;
SELECT-FROM-WHERE	π[a1, a2, a3, ...](σ[c](r1 × r2 × r3 × ...))	SELECT DISTINCT a1, a2, a3, ... FROM r1, r2, r3, ... WHERE c;
Inner Join	π[a1, a2, a3, ...]((r1 ⋈[c1] r2) ⋈[c2] r3)	SELECT DISTINCT a1, a2, a3, ... FROM r1 JOIN r2 ON c1 JOIN r3 ON c2;

Statement-Level Interface (SLI)

- Mix host language with SQL
  1. **Write** a program that mixes host language with SQL
  2. **Preprocess** the program using a preprocessor
  3. **Compile** the program into an exe code.
- Statement-Level Interfaces can serve Static OR Dynamic SQL.
  - Static SQL (PURPLE) → SQL query is fixed. Hard Coded. Variables declared beforehand
  - Dynamic SQL (GREEN) → SQL query is generated at run time. Suitable for unknown query struct @ compile time
  - Note: If there are multiple queries, each query will be prepared & processed independently

- EXEC SQL → Main keyword indicating statements to be preprocessed
- Connect to database with specified username, and to disconnect to releases resources

Basic Structure:
- **Declaration**
  - Declare variables
- **Connection**
  - Connect to DB
- **Execution**
  - Prepare queries
  - Execute queries
  - Operate on result
- **Deallocation**
  - Release resources

Declaration & Connection:

```
void main() {
    EXEC SQL BEGIN DECLARE SECTION;
        char  supplier[30], product[30];
        float price;
        char  query[500];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL CONNECT @localhost USER john;
    // some code that assigns values to
    // supplier, product and price
    EXEC SQL INSERT INTO
        Sells (supplName, prodName, price)
        VALUES (:supplier, :product, :price);
    //   query = "SELECT * FROM Sells";
    EXEC SQL EXECUTE IMMEDIATE :query;
    EXEC SQL DISCONNECT;
}
```

Declare Shared Variables

Can be used by host language; Get user input

Can be used by SQL (Auto convert between language & SQL); SQL query is fixed, only the argument changes. (Static SQL)

Call-Level Interface (CLI) Write in host language. Call functions from libraries through API. Libraries handle details

- **Declaration** → Declare variables, **Connection** → Connect to database,
- **Execution** → Execute queries + Operate on result, **Deallocation** → Release resources

No parameterized Query	Parameterized Query ** PREVENT SQL INJECTIONS

`# Open Cursor by executing query`
`my_query = 'SELECT...'`
`cursor.execute(my_query)`
	`myItemName = "my favourite item"`
	`# Execute Parameterized Statement`
	`cursor.execute("SELECT ... WHERE itemName = %s", (myItemName,))`
	`# DONT DO THIS: CAUSES SQL INJECTION`
	`cursor.execute("SELECT ... WHERE itemName = %s" % (myItemName,))`

Functions & Procedures

Return Types for Functions:
- <EMPTY> → One Existing Tuple
- SETOF → Set of Existing Tuples
- RECORD → One New Tuple
- SETOF RECORD → SET of New Tuples
- TABLE(param 1 TYPE1, ..., paramX TYPEX) → SET of New Tuples
- VOID → No Return Value *** or use Procedures

General Syntax for SQL Function

```
CREATE OR REPLACE FUNCTION <name>
    (<param> <type>, <param> <type>, ...)
RETURNS <type> AS $$
    <SQL logic code>
$$ LANGUAGE <language>;
```

Function Name

Function params & types

Return type of function

Main body of the SQL function

Enclose within $$, marks the start & end of query

Language used by SQL logic code

Stored Function vs Stored Procedure

Stored Function	Stored Procedure
• A function **returns a value** and is used in SQL expr. - SQL data types, Set of Existing tuples, - Set of new tuples - CREATE OR REPLACE FUNCTION (...) → SELECT(...) • Stored function MUST BE declared to return something (a special case is void) • Stored Function **cannot commit or rollback**	• A procedure is a block of SQL code that can execute multiple SQL statements. • Stored procedure **has no return value**, but can use out parameter. CREATE OR REPLACE PROCEDURE(...) →CALL(...) • Stored procedure **can commit or roll back**.

Cursor Flow
- A cursor enables us to access each individual row returned by a SELECT statement
- Declare → Open → Fetch → Check(repeat) → Close
  - FETCH [PRIOR | FIRST | LAST | ABSOLUTE N] [FROM] <cursor> INTO <var>
  - PRIOR → Fetch Previous row, FIRST/LAST → Fetch FIRST/LAST tuple, ABSOLUTE N → Fetch Nth tuple

Steps
1. **Declare** a cursor (With an associated SELECT query)
2. **Open** the cursor
3. **Fetch** a tuple from the cursor
4. Tuple **NOT FOUND**?
   - DO operation on the value otherwise
   - Construct value tuple by tuple
5. **Close** the cursor

- RETURN NEXT → Append 1 row to the output
- Need to use X_ (name with underscore). Else pgSQL will complain if the variables and columns have same names

```
CREATE OR REPLACE FUNCTION mark_gap()
RETURNS TABLE(name_ TEXT, mark_ INT, gap INT) AS $$

DECLARE
  curs CURSOR FOR (SELECT * FROM Scores ORDER BY Mark DESC);
  r    RECORD;   prev INT;

BEGIN
  prev := -1;   OPEN curs;

  LOOP
    FETCH curs INTO r;

    EXIT WHEN NOT FOUND;

    name_ := r.Name;   mark_ := r.Mark;
    IF prev >= 0 THEN gap := prev - mark_;
    ELSE gap := NULL;
    END IF;
    RETURN NEXT;
    prev := r.Mark;
  END LOOP;

  CLOSE curs;
END;

$$ LANGUAGE plpgsql
```

Trigger Timing (Effect of Return Value on the DB)

ACTION		NULL tuple	Non-NULL tuple t
BEFORE	INSERT	SKIP TG_OP! No tuple inserted	Tuple t will be inserted
	UPDATE	SKIP TG_OP! No tuple updated	Tuple t will be the updated tuple
	DELETE	SKIP TG_OP! No deletion performed	Deletion proceeds as normal
AFTER	INSERT UPDATE DELETE	Recall: RETURN value of AFTER does not affect the behavior on the DB. As such, NULL/NON-Values DOES NOT AFFECT/MATTER.	

Granularity & Timing
- INSTEAD OF → Only allowed on Row-Level Granularity
- AFTER/BEFORE → Allowed on BOTH Row & Statement

Timing	Row-Level	Statement-Level
AFTER	Tables	Tables & Views
BEFORE	Tables	Tables & Views
INSTEAD OF	VIEWS	-

```
CREATE [CONSTRAINT] TRIGGER trigger_name
{AFTER|BEFORE|INSTEAD OF} {event [or…]}
ON Table_Name          // INSERT|DELETE|UPDATE
[NOT DEFERRABLE | DEFERRABLE] {INITIALLY
DEFERRED/INITIALLY IMMEDIATE}
[FOR [EACH] {ROW | STATEMENT}]
[WHEN (condition)] //Should be a boolean clause
EXECUTE {FUNCTION | PROCEDURE} trig_fn_name();
```

Deferred Trigger
- Recall: Triggers can happen either at the end of statement/end of transaction.
- Deferred Trigger → Check consistency constraint only at the end of a transaction.
- Use keyword: "**CONSTAINT**" and "**DEFERRABLE**" **together** to indicate deferred trigger. (Often used tgt for AFTER)
- "**INITIALLY DEFERRED**" → By default, the trigger is deferred
- "**INITIALLY IMMEDIATE**" → By default, the trigger is NOT deferred
- **Deferred triggers ONLY work with AFTER and FOR EACH ROW (Row Level)**

Multiple Triggers
- There can be multiple triggers defined for the same event on the same table
- There need to be an **order of activation**
  - BEFORE statement-level triggers, BEFORE row-level triggers
  - AFTER row-level triggers, AFTER statement-level triggers
- Within each category, triggers are **activated in alphabetic order**
- If BEFORE row-level trigger returns NULL, then subsequent triggers on the same row are omitted.
- Raise notice → debug prints
- Raise Exception → Exception occurred. Transaction rollbacked/ aborted

Functional Dependencies → Reduce Data redundancy + Improve Data Integrity (No INSERT/DELETE/UPDATE anomalies)

4 Types of **Functional Dependencies** (given A & B) → 1. NULL (Not dependent), 2. A → B, 3. A ← B, 4. A ↔ B

Axioms	Implication	Rules	Implication
Axiom of Reflexivity	A set of attributes → A subset of the attributes	**Rule of Decomposition**	If A → BC, then A → B and A → C
Axiom of Augmentation	If A → B then AC → BC for any C	**Rule of Union**	If A → B and A → C, then A → BC
Axiom of Transitivity	If A → B and B →C, then A → B		

Computing Closures
- Given $A_1, A_2, ..., A_n$, the closure $\{A_1, A_2, ..., A_n\}^+$ can be computed as follows:
  1. Initialize the closure to $\{A_1, A_2, ..., A_n\}$
  2. If there is an FD: $A_i, A_j, ..., A_m \rightarrow B$, such that $A_i, A_j, ..., A_m$ are all in the closure, then put B into the closure
  3. Repeat step 2, until we cannot find any new attribute to put into the closure

Closure & Functional Dependency

Prove Existence	Prove Non-Existence
• To prove that X → Y holds, we only need to show that $\{X\}^+$ contains Y • Example: AB→C, AD→E, B→D, AF→B; Prove that AF→D - $\{AF\}^+ = \{AFBCDE\}$, which contains D - Therefore, AF→D holds	• To prove that X → Y does not hold, we only need to show that $\{X\}^+$ does not contain Y • Example: AB→C, AD→E, B→D, AF→B; Prove that AD→F does not hold - $\{AD\}^+ = \{ADE\}$, which does not contain F - Therefore, AD→F does not hold

3NF	BCNF
For any non-trivial and decomposed FD: • Either the LHS is a super key, OR • RHS is a prime attribute	For any non-trivial and decomposed FD: • The LHS is a super key.
3NF is more "lenient" than BCNF. Satisfying BCNF → satisfying 3NF, & Violating 3NF → violating BCNF	
Decomposition	
A 3NF decomposition has **only one split,** which divides the table into two or more parts	A BCNF decomposition may perform one/more binary splits, each of which divides a table into two (recursive)

How to determine if FD is Non-trivial and Decomposed FD?
1. Consider all attribute subset in R
2. Compute the closure of each subset
3. From each closure, remove the "trivial" attributes
4. Derive non-trivial and decomposed FDs from each closure

Check Dependency (If more than 2 Decompositions, do for all)
- Step 1: Find projections of Σ on each R_i
- Step 2: Show that $(\Sigma | R_1 \cup \Sigma | R_2)$ can be derived from Σ
- Step 3: Show that Σ can be derived from $(\Sigma | R_1 \cup \Sigma | R_2)$

**Does not depend on SK.

SuperKey (SK)	• A set of attributes in a table that decides all other attributes. • A set of attributes that can uniquely identify a tuple in a relation
Candidate Key (Key)	• A superkey that is minimal • A minimal superkey (no proper subset is a superkey) - A relation can have multiple candidate keys - If we remove any attribute from the superkey, it will not be a superkey anymore
Primary Key	The candidate key chosen by the database designer as the main identifier
Prime Attribute	• **Prime attribute**: An attribute that is part of any candidate key • **Non-prime attribute**: An attribute not part of any candidate key

Tips & Trick to find Keys
- **Check the smallest attributes** first (e.g 1 element) → {Z}+ → {A…Z}, any superset of Z is also a key, BUT not minimal
- **Check attribute's appearance on RHS** → If Z does not appear on RHS of any FDs, then Z MUST be in every key

BCNF: A table R is in BCNF, if **every non-trivial and decomposed FD has a superkey** on its **left hand side**

Trivial FDs	In Trivial FD, a dependent is always a subset of the determinant. i.e. If X → Y and Y is the subset of X, then it is called trivial FD. E.g AB → A
Non-Trivial FDs	In **Non-trivial FD**, the dependent is strictly not a subset of the determinant. i.e. If **X → Y** and **Y is not a subset of X**, then it is called Non-trivial FD. E.g A → B

Non Trivial but not completely trivial:
- Elements in LHS & RHS
- E.g **AB → BCD**
  NOT: AB → B

- 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.
  - All attributes B can depend ONLY on the superkeys. Any dependency on NON-SUPERKEYS → Violates BCNF
- BCNF requires that **every non-trivial, left-irreducible FD (X → Y)** must have **X as a superkey**.
  - If X is not a superkey, then:
    - The same X can appear multiple times in the relation (since X alone doesn't uniquely identify a row).
    - Each time X repeats, Y must also repeat (due to X → Y).
    - This leads to redundancy (duplicate data).

BCNF Properties
- No update or deletion or insertion anomalies
- Small redundancy
- The original table can always be reconstructed from the decomposed tables.
- Dependencies may not be preserved (BAD)

Simplified Algorithm for BCNF Check
1. Compute the closure of each attribute subset
2. Check if there is a closure $\{A_1A_2...A_k\}^+$, such that:
   - The closure contains some attribute not in $\{A_1A_2...A_k\}$
   - The closure does not contain all attributes in the table
   - i.e., a "**more but not all**" closure
3. If such a closure exist, then R is NOT in BCNF.

Algorithm for BCNF Table Decomposition **Recursive, BCNF Decomposition table is NOT UNIQUE

Input: a table R ** A table with only 2 attributes MUST be in BCNF
1. Find a subset X of the attributes in R, such that its closure $\{X\}^+$ (i) contains more attributes than X does, but (ii) does not contain all attributes in R (Find a violator, more but not all)
2. Decompose R into two tables R_1 and R_2, such that
   - R_1 contains all attributes in $\{X\}^+$
   - R_2 contains all attributes in X as well as the attributes not in $\{X\}^+$
3. If R_1 is not in BCNF, further decompose R_1;
   If R_2 is not in BCNF, further decompose R_2
4. REPEAT if necessary

- In general, **each decomposition step removes at least one BCNF violation**
- Recursive decomposition → **all violations will be removed in the end**

Lossless Join Decomposition
- Say we decompose a table R into **two** tables R_1 and R_2
- The decomposition guarantees lossless join whenever the **common attributes in R_1 & R_2 constitute a superkey of R_1 or R_2**
- Example: R(A, B, C) → R_1(A, **B**) & R_2 (**B,** C), with **B** = SK of R_2 | R(A, B, C, D) → R_1(A, **B, C**) & R_2 (**B, C,** D), with **BC** = SK of R_1

Why BCNF guarantees lossless join?
- Decompose R into two tables R_1 and R_2, such that
  - R_1 contains all attributes in $\{X\}^+$ & R_2 contains all attr in X & attributes not in $\{X\}^+$
- X is the set of common attributes between R_1 and R_2. X is a superkey of R_1

Check Lossless-Join (Given R_1 & R_2)
1. Check $R_1 \cap R_2 \rightarrow R_1$
2. Check $R_1 \cap R_2 \rightarrow R_2$
3. Check if $R_1 \cap R_2$ is a SK for R_1 OR R_2

** If R_i don't have SK → Not Lossless
Note: If > 2 Decompositions, we need to find a sequence of $R_1, R_2, ..., R_n$ decomposition tree that creates a lossless join. If ∅, then it is NOT lossless. NOT pairwise comparison

Definition: Dependency Preservation
- Let S be the given set of FDs on the original table.
- Let S' be the set of FDs on the decomposed tables
- We say that the decomposition preserves all FDs IFF S and S' are equivalent. i.e
  - Every FD in S' can be derived from S.
  - Every FD in S can be derived from S'. I.e Decomposition: $S \leftrightarrow S'$

Check for 3NF
Input: a table R
1. Derive the keys of R
2. For each given FD, check if
   - The **left hand side is a superkey**, or
   - Each attribute on the **right hand side is a prime attribute**
3. If all given FDs satisfy this condition, then R is in 3NF

Minimal Basis/Cover Algorithm
1. Transform the FDs, (decompose) so that each RHS contains ONLY 1 attribute
2. Remove redundant attributes on the LHS of each FD.
3. Remove redundant FDs.
(Don't swap step 2 and 3. This might cause inaccurate results.)

Minimal Basis
- Given a set S of FDs, the minimal basis of S is a simplified version of S (AKA minimal cover of S)
- A set of FDs is a minimal basis/minimal cover, if it satisfies ALL 4 conditions:
  1. **Every FD in the minimal basis** can be **derived from S**, and **vice versa.** **equivalence
  2. **Every FD** in the minimal basis is a **non-trivial and decomposed FD**
  3. **No** FD in the minimal basis is **redundant**.
     - if any FD is removed from M, then some FD in S cannot be derived from M
  4. For each FD in the minimal basis, **none of the attributes on the left hand side is redundant**.
     - if we remove an attribute from its left hand side, then the FD cannot be derived from S

3NF Decomposition Algorithm: Given: A table NOT in 3NF. **Objective**: Decompose it into smaller tables that are in 3NF
1. Derive a minimal basis of S. [E.g min_basis = {AD→C, CD→B} → R(A,B,C,D) → R_1(ADC), R_2(BCD)]
2. In the minimal basis, combine the FDs whose LHS are the same
3. Create a table for each FD remained
4. If none of the tables contains a key of the original table R, create a table that contains a key of R (any key would do)
   - Check if any of the tables contain a key for R; else; create a table that contains a key for R
   - i.e Adding Key for Lossless Join. **Note: This step is essential to ensure lossless join decomposition.**
5. Remove subsumed/redundant tables
   - In general, we remove a table R', if all of its attributes are contained in another table R''
   - E.g R_1(A,B,C), R_2(A,C), we can remove R_2(A,C), since A & C are already in R_1(A,B,C)

How to Find Keys?
1. Start with any superkey
2. Remove attributes one by one while ensuring the remaining set is still a superkey
3. Stop when no more attributes can be removed without losing the superkey property

OR
- Step 1: List All Possible Subsets
- Step 2: Compute Attribute Closures
- Step 3: Identify All Superkeys
- Step 4: Extract Minimal Superkeys (Candidate Keys)