CS2102 Cheatsheet AY24/25 ——@Jin Hang

Database Management Systems
- Most since 1970's are relational
- Support SQL for the definition, manipulation, query, control

Architecture
- Client-Server Architecture: Distributed and concurrent access
- Three-Tier Architecture: presentation tier (clients), business logic tier (application server), data tier (database server).

Relational Model (RDBMS organise data in **tables**)
- Relations have a name and attributes. Attributes have a name.
- Rows or **records** have **fields** corresponding to the columns.
- Columns or fields have a **domain** which is a type to the extension of which is added the possibility of a NULL value.

Consistent State complies with the business rules as defined by structural constraints and integrity constraints in the schema.
- violated → operation/transaction is aborted and rolled back
- Concurrency Control and Recovery Mechanisms should ensure atomicity, consistency, isolation, and durability of transactions
- Immediate: they are checked after each operation
- Most sys. only allow certain constraints to be deferred.

Integrity Constraints
- NOT NULL: use DEFAULT to address Implicit NULL
- PRIMARY KEY
- UNIQUE: cannot contain two records with the same value
- FOREIGN KEY
- CHECK followed by the SQL condition in parenthesis.

Primary key uniquely identifies a record. Each table at most one
- can be one column or a combination of columns.
- is equivalent to UNIQUE and NOT NULL.

REFERENCES Required to be primary key for portable
- Check for the existence of the set of values in referenced table.
- Omit the check if any value in the set to be checked is NULL.
- Prevent Deletion

Wildcards: The asterisk (i.e., * symbol) is a shorthand indicating that all the column names, in order of the CREATE TABLE declaration, should be considered in the SELECT clause.

```
SELECT * FROM games WHERE version = '2.0' ORDER BY price;
```

DISTINCT
- It often gives a sorted result but this is not guaranteed.
- Only apply whole selections cannot apply on a specific column

Combining Constructs: Not all combinations make sense. Both DISTINCT and ORDER BY conceptually involve sorting and ORDER BY is **applied before** DISTINCT. SQL does not know which row to remove.

WHERE: filter rows on a Boolean condition.

```
SELECT first_name, last_name FROM customers
WHERE country IN ('Singapore', 'Indonesia')
  AND (dob BETWEEN '2000-01-01' AND '2000-12-01'
      OR since >= '2016-12-01')
  AND last_name LIKE 'B%';
```

AS is a renaming operator.

```
SELECT name||' '||version AS game, ROUND(price * 1.09, 2)
FROM games  // || is a concatenation operator.
WHERE price * 0.09 >= 0.3;
```

Logic and NULL "`SELECT FROM WHERE <condition>`" returns results when the `<condition>` is True
- `NOT(X<>a AND X<>b)` ⇔ `X=a OR X=b`
- Every domain has the additional value: the NULL value. It generally (but not always) has the meaning of "**unknown**". `10 + NULL`, `0 * NULL`, `X > NULL` is null! (even if X is NULL)
- `SELECT col1, col2 FROM t WHERE col2 <> NULL;` ⇒ Nothing!
- `COALESCE(x1, x2, x3, ...)` returns the **first non-NULL**

```
SELECT column1, column2, COALESCE(column2, 0) AS col2
FROM example WHERE column2 IS NOT NULL; // NULL -> 0
CASE WHEN column2 is NULL THEN 0 ELSE column2 END // Equal
```

- `COUNT(*)` counts NULL values.
- `COUNT/AVG/MAX/MIN(att)` eliminate NULL values.

P	Q	P AND Q	P OR Q	NOT P
True	True	True	True	False
True	False	False	True	False
True	Unknown	Unknown	True	False
False	True	False	True	True
False	False	False	False	True
False	Unknown	False	Unknown	True
Unknown	True	Unknown	True	Unknown
Unknown	False	False	Unknown	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

- `NULL IS NULL` ⇒ [T]
- `1 IS NULL` ⇒ [F]
- `1 IS NOT NULL` ⇒ [T]
- `NULL IS NOT NULL` ⇒ [F]

Multiple Table
- **Cartesian product** of tables containing all the columns and all possible combinations of the rows of the three tables.
- Comma in the `FROM` ⇔ keyword `CROSS JOIN`.
- Join tables according to **primary** and **foreign** keys
- Add conditions that referencing attributes = referenced attributes in `WHERE`
- ×table var. ⇒ sys. cannot understand which column is which.

Aggregation Functions `COUNT, SUM, MAX, MIN, AVG, STDDEV`
- `SELECT COUNT(ALL c.country) FROM customers c;` ALL is default and often omitted.
- `SELECT COUNT(DISTINCT c.country) FROM customers c;`
- `TRUNC(AVG(g.price), 3) AS avg` display 3 decimal places

SELECT MIN(A) FROM R;	Minimum non-null value in A
SELECT MAX(A) FROM R;	Maximum non-null value in A
SELECT AVG(A) FROM R;	Average of non-null values in A
SELECT SUM(A) FROM R;	Sum of non-null values in A
SELECT COUNT(A) FROM R;	Count of non-null values in A
SELECT COUNT(*) FROM R;	Count of rows in R
SELECT AVG(DISTINCT A) FROM R;	Average of distinct non-null values in A
SELECT SUM(DISTINCT A) FROM R;	Sum of distinct non-null values in A
SELECT COUNT(DISTINCT A) FROM R;	Count of distinct non-null values in A

Grouping `GROUP BY` creates logical groups of records that have the same values for the specified fields.
- Groups are formed **before** computing the aggregate functions and **after** rows have been filtered by the `WHERE` clause.
- The aggregation functions are calculated for each logical group.
- `SELECT c.country, COUNT(*) FROM customers c;` ⇒ ERROR
- It is recommended (and required per SQL standard) to include attributes projected in `SELECT` clause by `GROUP BY` clause.

```
SELECT c.country, EXTRACT(YEAR FROM c.since) AS regyear
.... // Extract year part and rename to regyear
GROUP BY c.country, regyear ORDER BY regyear ASC;
```

`Having` performed after `GROUP BY`. - **Aggregate Condition**
- Aggregate functions in conditions cannot be used in `WHERE`
- Only use aggr. func, columns listed in `GROUP BY` and subqueries.

Chain `FROM->ON->OUTER JOIN->WHERE->GROUP BY-> AGGREGATE-> HAVING->SELECT->DISTINCT->ORDER BY->LIMIT`
- `HAVING` is used to filter groups after aggregation.
- `WHERE` is used to filter rows before aggregation

Inner Join: no added expressiveness or performance
- `JOIN` is synonymous with `INNER JOIN`.
- Do NOT recommend either as `CROSS JOIN` (or comma) is typically easier to read and **will be optimized by DBMS**.

Nature Join used when same name columns have same meaning
- Joins rows have same values for columns with the same name
- Prints only one of the two columns

Outer Join Try to avoid as they introduce `NULL` values.
- **Further restriction** should be on `WHERE` and not `ON` clause.
- Conditions in the `ON` clause determines which rows are dangling.

Left Outer Join Returns all records from the left table and matching records from the right table; unmatched records from the right table are `NULL`. [Pads **Right** Table with `NULL`]

```
SELECT c.customerid, d.customerid, d.name, d.version
FROM customers c LEFT OUTER JOIN downloads d ON c.
     customerid = d.customerid;
```

Right Outer Join: Pads **Left** Table with `NULL`
Full Outer Join pads missing values with NULL for both tables.
You can survive with just cross and left join. Just see whether you want NULL or not.
Union-Compatible: Same number of columns with the same domain in the same order ⇒ use `UNION`, `INTERSECT`, or `EXCEPT`.
- union-compatible ≠ it is meaningful
- Need not to have same column name

Nested Query

Visible Change `VIEW` change when the base table change. views are unmaterialized and materialized views need to be refreshed.
- Hide data and/or complexity from users
- Logical data independence

```
CREATE VIEW overdue AS
SELECT b.title FROM books b, users u, ... WHERE ...
SELECT * FROM overdue; -- VIEW
```

```
WITH singapore_customer AS (
  SELECT * FROM customers c
  WHERE c.country = 'Singapore'
) SELECT cs. last_name, d. name
FROM singapore_customer cs, downloads d
WHERE cs. customerid = d. customerid;
```

```
SELECT cs.last_name, d. name
FROM (
  SELECT * FROM customers c WHERE c.country = 'Singapore'
) AS cs, downloads d
WHERE cs.customerid = d.customerid;
```

```
SELECT (SELECT COUNT(*) FROM customers c
        WHERE c.country = 'Singapore');
```

Store Procedure

Static SQL: Query fixed; Only argument changes

```
EXEC SQL INSERT INTO Sells (supplName, prodName, price)
                    VALUES (:supplier, :product, :price);
```

Dynamic SQL: Query is generated at run-time

```
query = "SELECT * FROM Sells"
EXEC SQL EXECUTE IMMEDIATE :query;
```

Prepared SQL: Prepare once; Execute multiple times; Use placeholder "?"

```
void main() {
  EXEC SQL BEGIN DECLARE SECTION;
    const char *query = "INSERT INTO test VALUES(?, ?);";
  EXEC SQL END DECLARE SECTION;
  EXEC SQL CONNECT @localhost USER john;
  // Parsed, compiled by database only ONCE
  EXEC SQL PREPARE stmt FROM :query;
  EXEC SQL EXECUTE stmt USING 42, 'foobar';
  EXEC SQL DEALLOCATE PREPARE stmt;
  EXEC SQL DISCONNECT;
}
```

Call-Level Interface: Code is written only in host language; Use of API call to run SQL queries.

```
product = "some product"
conn = psycopy.connect("host=...")
cursor = conn.cursor() // Create cursor
cursor.execute("SELECT supplName from Sells where prodName
      = %s", (product,)) // parameterized statement
while True:
    row = cursor.fetchone()
    if row is None: break
    print(f">>> Supplier: {row[0]})
cursor.close(); conn.commit(); conn.close(); // clean up
```

Function	Procedure
• Must return something (special case is void) • Cannot roll back • Use SELECT to call	• No return, but **can** use **OUT** parameter. • Can commit or roll back. • CALL <procedure name>()

Functions

```
CREATE OR REPLACE FUNCTION <name> (<param> <type>, ...)
RETURNS <type> AS $$ ...
$$ LANGUAGE <language>;
SELECT * FROM <name> (<param> <type>, ...);
```

- RECORD: Single tuple → must have **at least** two OUT parameters
- TABLE (x INT , y INT) AS $$: Multiple tuples
- SETOF: A set of tuples
- VOID: No return value

```
# returns a tuple (Mark , Count)
CREATE OR REPLACE FUNCTION CountGradeStudents
    (IN Grade CHAR (1) OUT Mark CHAR (1) , OUT Count INT)
RETURNS RECORD AS $$
    SELECT Grade, COUNT (*) FROM Scores
    WHERE convert(Mark) = Grade;
$$ LANGUAGE sql;
SELECT CountGradeStudents(90);
```

```
CREATE OR REPLACE FUNCTION topMarkCount
(OUT Mark INT, OUT Count INT) RETURNS RECORD AS $$ ...
SELECT * FROM topMarkCount() -- No input parameter
```

```
... median() RETURNS setof Scores AS $$
DECLARE
  curs CURSOR FOR(SELECT * FROM Score ORDER BY Mark DESC);
  r RECORD; num_student INT;
BEGIN
  OPEN curs;
  SELECT COUNT(*) INTO num_student FROM Scores;
  IF num_student%2 = 1 THEN ...
  ELSE
    FETCH ABSOLUTE num_student/2 FROM curs INTO r;
    Name := r.Name; Mark := r.Mark; RETURN NEXT;
    FETCH ABSOLUTE num_student/2+1 FROM curs INTO r;
    Name := r.Name; Mark := r.Mark; RETURN NEXT;
  END IF;
  CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

SQL Procedures: No return value, but can use out parameter

```
CREATE OR REPLACE PROCEDURE transfer (frAcc TEXT, toAcc
     TEXT, amount INT) AS $$
  UPDATE Acct SET balance=balance-amount WHERE name=frAcc;
  UPDATE Acct SET balance=balance+amount WHERE name=toAcc;
$$ LANGUAGE sql;
CALL transfer('Alice','Bob',100);
```

Control Structures:

```
DECLARE temp INT; BEGIN temp := val1; END;
```

```
s := 0; temp := 1;
WHILE temp <= x LOOP
   s := s + temp; temp := temp + 1;
END LOOP;
```

```
LOOP # repetition, while (true) { if (temp > x) break; }
   EXIT WHEN temp > x;
   s := s + temp; temp := temp + 1;
END LOOP;
```

```
FOR row IN SELECT * FROM Scores ORDER BY Mark DESC
LOOP
   ...
   RETURN NEXT;
END LOOP;
```

Cursor: Access each individual row returned by a SELECT stat. Declare → Open → Fetch → Check (repeat) → Close

```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE(name_ TEXT, mark_ INT, gap INT) AS $$
DECLARE
  -- Associated with a SELECT statement at declaration
  curs CURSOR FOR(SELECT * FROM Score ORDER BY Mark DESC);
  r RECORD; prev INT;
BEGIN
  prev := -1; OPEN curs;
  LOOP
    FETCH curs INTO r; -- FETCH a tuple from the cursor
    EXIT WHEN NOT FOUND;
    name_ := r.Name; mark_ := r.Mark;
    IF prev >= 0 THEN gap := prev -mark_;
    ELSE gap := NULL;
    END IF;
    RETURN NEXT;  -- Append current row to RECORD to output
    prev := r.Mark; -- still current row
  END LOOP; CLOSE curs;
END;
$$ LANGUAGE plpgsql;
```

- FETCH PRIOR FROM curs INTO r; - Fetch from previous row
- FETCH FIRST/LAST FROM curs INTO r;
- FETCH ABSOLUTE 3 FROM curs INTO r; - Fetch the 3rd tuple

SQL Injection Attacks: A class of attacks on dynamic SQL

```
query += "SELECT * FROM T WHERE Name = '" + name + "';";
EXEC SQL EXECUTE IMMEDIATE :query;
```

- Malicious Usage: "'; DROP TABLE T; SELECT '"
- Protect: Use prepares statements (compiled when prepared) → At runtime, anything in name is treated as a string

```
string query = "SELECT * FROM T WHERE Name = ?;";
EXEC SQL PREPARE stmt FROM :query;
EXEC SQL EXECUTE stmt USING :name;
```

Trigger event-condition-action (ECA) rule

```
-- TRIGGER
CREATE TRIGGER <trigger_name> <timing> <event> ON <table>
FOR EACH <granularity> WHEN (<condition>)
EXECUTE FUNCTION <function_name>();
-- TRIGGER FUNCTION
CREATE OR REPLACE FUNCTION <name>() RETURNS TRIGGER AS $$
BEGIN
<code>
END;
$$ LANGUAGE plpgsql
```

Trigger Function: Can only RETURNS TRIGGER

- TG_OP - operation (INSERT/DELETE/UPDATE)
- TG_TABLE_NAME - refers to table name
- NEW - new row inserted, only accessible by trigger functions
- OLD The modified row before the triggering event

<event>

- INSERT ON table / DELETE ON table
- UPDATE ON table / UPDATE OF column ON table

<granularity>

- FOR EACH ROW (that is modified) - you can insert into multiple rows with a single INSERT statement
- FOR EACH STATEMENT (that performs the modification) **ignores** the values returned by the trigger functions
  - RETURN NULL **will not** make DB omit subsequent operation
  - Use RAISE EXCEPTION to abort current transaction.
  - Executes the trigger function only once

```
... FUNCTION show_warning() RETURNS TRIGGER AS $$
BEGIN
    RAISE NOTICE 'You are ..'; -- Throw Error and Roll Back
    RETURN NULL; -- won't make DB omit subsequent operation
    -- Cannot omit RETURN.. for Trigger, otherwise throw E
END;
$$ LANGUAGE plpgsql;
```

<timing>

- AFTER or BEFORE (the triggering event)
- INSTEAD OF (the triggering event on views) - can **only** be defined on VIEW, not be done on statement-level

```
DELETE FROM Students WHERE Name = 'Alice';
DELETE FROM Scores WHERE Name = 'Alice'; -- Auto Translate
```

Views return **aggregate functions** × automatically updatable

```
CREATE VIEW Top AS SELECT MAX(Mark) AS Mark FROM Scores;
UPDATE Top SET Mark = 80;
```

```
... FUNCTION update_top_mark() RETURNS TRIGGER AS $$
BEGIN
  UPDATE Scores SET Mark = NEW.Mark WHERE Mark = OLD.Mark;
  RETURN NEW; -- Trigger works in both NULL and NEW
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER modify_top_mark
INSTEAD OF UPDATE ON Top_Marks -- Trigger on VIEW
FOR EACH ROW EXECUTE FUNCTION update_top_mark();
```

Effect	NULL tuple	non-NULL tuple t
`BEFORE INSERT`	No tuple inserted	Tuple t will be inserted
`BEFORE UPDATE`	No tuple updated	Tuple t will be the updated tuple
`BEFORE DELETE`	No deletion performed	Deletion proceeds as normal
`AFTER *`	Does not matter	

```
... FUNCTION give_adi_full_mark() RETURNS TRIGGER AS $$
BEGIN -- OLD is a row with all attributes to be NULL
    IF (NEW.Name = 'Adi') THEN
        -- Name: NULL->Adi, Mark: NULL->100
        OLD.Name := 'Adi'; OLD.Mark := 100;
    END IF;
    -- Since OLD has some attributes NOT NULL, insert OLD
    -- If NEW.Name != Adi, OLD is ignored -> not insert
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Return Values:

- `NULL` - ignore all operations on current row
- `NOT NULL` - signals the database to proceed as normal

```
... FUNCTION update_max() RETURNS TRIGGER AS $$
BEGIN
  UPDATE Scores SET Mark = NEW.Mark WHERE Mark = OLD.Mark;
  RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

Triggers Condition: `WHEN`

```
FOR EACH ROW WHEN (NEW.Name = 'Adi') EXECUTE FUNCTION f();
```

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

Deferred Trigger: happen at the end of statement/transaction

```
CREATE CONSTRAINT TRIGGER balance_check
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY DEFERRED
FOR EACH ROW EXECUTE FUNCTION f();
```

- Operation with multi. statements may leave DB in an intermediate inconsistent state
- Only work with `AFTER` and `FOR EACH ROW`
- `INITIALLY IMMEDIATE` - not deferred by default ⇒ `SET CONSTRAINTS f DEFERRED;` in transaction to defer

```
BEGIN TRANSACTION;
  SET CONSTRAINTS balance_check DEFERRED;
  UPDATE ...
COMMIT;
```

Relational Algebra: Imperative query language.

- No duplicate row in relational algebra
- Relational algebra → based on relations → based on set.

Propositional Logic: Defined by the truth tables of connectives.

- **Equivalent** iff. they have the **same** truth table.

Standard Operator: $=, \neq, >, <, \geq, \leq$

- Lexicographically Comparison for text

Selection: $\sigma_{[c]}(R)$ selects all rows from R that satisfies condition c

- `SELECT * FROM R WHERE c`

Projection: $\pi_{[l]}(R)$ keeps col. in ordered list l with same order

- $\pi_{[cname]}(likes) \Leftrightarrow$ `SELECT DISTINCT l.cname FROM likes l;`

Rename: $\rho(R_1, R_2)$ rename the relation R_1 to R_2

- Not create a new relation in the hard disk, simply refer as R_2
- $\rho(R_1, R_2(A_1 \to B_1))$ rename relation and some of its attributes
- $\pi_{[r.attr]}(\sigma_{[c]}(\rho(rel, r))) \Leftrightarrow$ `SELECT DISTINCT r.attr FROM rel r WHERE c;`

Set Operators: Unoin $\cup$, Intersect $\cap$, Except $-$

- Two relations must be union-compatible (same column types)
- $Q_1 := \ldots, Q_2 := \ldots, Q_1 \cup Q_2$

Cross Product: $R_1 \times R_2$ 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.
$\pi_{[c.cname, r.rname]}(\sigma_{[c.area=r.area]}(\rho(cust, c) \times \rho(rest, r))) \Leftrightarrow$ `SELECT c.cname, r.rname FROM cust c, rest r WHERE ...;`

Join

- **Natural Join:** $R_1 \bowtie R_2$
- Natural Left/Right/Full Outer Join: $R \bowtie^{-} / \bowtie^{-}_{-} / {}^{-}_{-}\bowtie^{-} S$
- $R_1 \bowtie_{[c]} R_2 \Leftrightarrow \sigma_{[c]}(R_1 \times R_2)$.

ER Diagram

- Relationships can have attributes but they are distinguished by their participating entities.
- Weak Entities naturally has a partial key
  ⇒ Partial key uniquely identifies an instance within its related dominant entity but not globally
  ⇒ Relationship set is now defined by a combination of both the dominant entity's key and the weak entity's partial key.

Candidate Keys: All sets of attributes that uniquely identify the entities. But there must be **at least one** primary keys

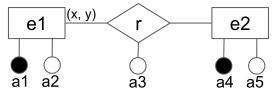
Schema Translation is **simple** (3 rules + 3 exceptions), cannot translate other cardinalities (e.g., $(1, n)$).

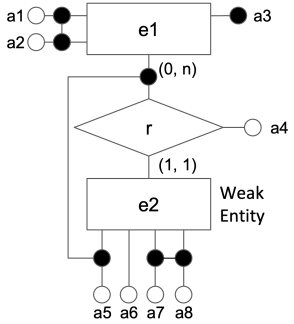
- Value sets are mapped to **domains**.
  ER attributes → attributes of relations with meaningful type.
- Entity sets are mapped to relations.
  CKs → PKs and `UNIQUE NOT NULL` (Should be at least one PK)
- Relationship sets are mapped to relations.
  The keys are the keys of the participating entities.

Aggregate is simply a relationship set, but can be used as entity set ⇒ keys can be referenced by another relationship set.

Exceptions in Translation

- $(0, 1)$: PK(r)=$a1$, $a3$ `NOT NULL`
- $(1, 1)$: Merge $e2$ and r
  PK(r)=PK($e1$)=$a1$
  $a4$ set `NOT NULL` as FK
- $(0, n)$: $\{a1, a4\} \to \{a3\}$
- Generall $(0, n)$, PK(r)=Keys of $e1, e2$=$\{a1, a4\}$.
  Also Set $a1$, $a4$ as Foreign Key
- Weak Entity: $e1$ is dominant entity, $e2$ is weak entity





Merge e2 and r
Cardinality:
Given N e2 → e1: $[1, N]$
Given N e1 → e2: $[0, N]$
Candidate Key:
$e1 = \{\{a3\}, \{a1, a2\}\}$
$e2, r = \{\{a3, a5\}, \{a1, a2, a5\}, \{a7, a8\}\}$
Functional Dependency:
$\{a1, a2\} \to \{a3\}$
$\{a3\} \to \{a1, a2\}$
$\{a3, a5\} \to \{a4, a6, a7, a8\}$
$\{a1, a2, a5\} \to \{a4, a6, a7, a8\}$
$\{a7, a8\} \to \{a3, a4, a5, a6\}$

Principle Ensure identities can **uniquely identify** entity set.
Ensure **minimum/maximum cardinality** is satisfied

Normal Form: minimum requirements to reduce data redundancy, and improve data integrity

- No Update/Deletion/Insertion anomalies

Functional Dependencies Let $A_1, A_2, \ldots, A_m, B_1, B_2, \ldots, B_n$ be some attributes. $\{A_1 \ldots A_m\} \to \{B_1 \ldots B_n\} \Leftrightarrow \{A_1 \ldots A_m\}$ **uniquely identifies** $\{B_1 \ldots B_n\}$

- An FD may hold on one table but does not hold on another

Armstrong's Axioms

- Reflexivity: Set → Subset of attributes ($\{A, B\} \to \{A\}$)
- Augmentation: if $\{A\} \to \{B\}$, then $\forall C$, $\{AC\} \to \{BC\}$
- Transitivity: if $\{A\} \to \{B\}$ and $\{B\} \to \{C\}$, then $\{A\} \to \{C\}$
- Decomposition: if $\{A\} \to \{BC\}$ then $\{A\} \to \{B\} \wedge \{A\} \to \{C\}$
- Union: if $\{A\} \to \{B\} \wedge \{A\} \to \{C\}$, then $\{A\} \to \{BC\}$

Closure: $\{A_1 A_2 \ldots A_m\}^+ = B_1 B_2 \ldots B_n$

- Prove $X \to Y$ holds or not ⇔ Show $\{X\}^+$ contains Y or not

Super Key: A set of attributes that decides all other attributes

Key: A superkey that is minimal.

- Whether or not a table T has redundancy and anomalies would partially depend on what the keys of T are

Trick

- Always check small attribute sets first
- If an attribute does not appear in the RHS of any FD, then it must be in every key

Prime Attributes: Attribute appears in a key

Decomposed FD: FD whose RHS has only one attribute ($\{X_1 \ldots X_m\} \to Y$) a non-decomposed FD can always be transformed into an equivalent set of decomposed FDs

Non-Trivial: Consider FD: $X \to Y$, X is

- Trivial if $Y \subseteq X$.
- Non-trivial if $Y \nsubseteq X$.
- Completely non-trivial if $Y \neq \varnothing$ and $X \cap Y = \varnothing$.

Find Non-trivial and Decomposed FD

1. Consider all attribute subsets 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

BCNF: if every non-trivial and decomposed FD of table R has a superkey on its LHS

- For $\{A_1 \ldots A_m\} \to B$, $\{A_1 \ldots A_m\}$ must be a superkey.
- All attributes B can depend only on superkeys
- If B depends on non-superkey $C_1 C_2 \ldots C_n$. Since $C_1 C_2 \ldots C_n$ is not a superkey ⇒ appear multiple times in the table ⇒ same B would appear multiple times in the table ⇒ **redundancy**

Simplified BCNF Check

1. Compute the closure of each attribute subset
2. Check if ∃ more but not all closure ⇒ Violate BCNF

Normalization: Decompose R into smaller tables

BCNF Decomposition

- BCNF decomposition of a table may not be unique
- If a table has **only two** attributes, then it must be in BCNF
- Each decomposition step removes at least one BCNF violation
- Dependencies **may not** be preserved
  $R = \{A, B, C\}, \sum = \{AB \to C, C \to B\} \to \{BC, AC\}$

1. Find a subset X of attributes, s.t. $\{X\}^+$ is more but not all
2. Decompose R into two tables R_1 and R_2, such that
   - R_1 contains everything in$\{X\}^+$
   - R_2 all attributes in X as well as attributes not in$\{X\}^+$
   - Suppose $X \to Y$ violates BCNF, therefore it is no longer a (violation on R_1, FD on R_2)
3. Further decompose R_i if not in BCNF
   - If we don't know what FDs are there on R_i
   - Derive the closures on $R \to$ **project** them onto R_i

Closure Projection: Project each closure of R onto R_i by removing those attributes that do not appear in R_i

Lossless Join Decomposition: Decomposition guarantees lossless join, whenever the **common attributes** in R_1 and R_2 constitute a superkey of R_1 or R_2

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
- Prove $S \leftrightarrow S'$

Third Normal Form: iff. $\forall$ non-trivial and decomposed FD, Either the LHS is a superkey Or RHS is a prime attribute.

- More lenient than BCNF. BCNF $\rightarrow$ 3NF; $\neg$ 3NF $\rightarrow \neg$BCNF

3NF Decomposition: Given table R, and a set S of FDs

- One split divides the table into two or more parts
- Has update and deletion anomalies in some rare cases

1. Derive a minimal basis of S
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 [To ensure lossless join decomposition]
5. Remove subsumed tables. E.g. $R_1 = (A, C)$, $R_2 = (C, D, E)$, $R_3 = (C, E) \rightarrow$ Remove R_3 since $R_3 \subset R_2$

Minimal Basis/Cover

- Every FD in minimal basis can be derived from S, and vice versa
- Non-trivial and decomposed FD.
- No FD in the minimal basis is redundant.
- None of the attributes on the LHS is redundant

1. Transform the FDs s.t. each RHS contains only one attribute
2. Remove redundant attributes on LHS of each FD
   - For $X_1X_2 \rightarrow Y$, remove another attr. if $Y \in \{X_1\}^+$, $\{X_2\}^+$
3. Remove redundant FD
   - For $X \rightarrow Y$, check $Y \in \{X\}^+$ after removing it from FD

Canonical Cover: Compact Minimal Cover. No different FD with same LHS $\Rightarrow$ Merger by same LHS

BCNF or 3NF

BCNF	3NF
• Small redundancy • Lossless join property • But may not preserve all • One or more binary splits to two table each time	• Not as strict as BCNF • Lossless join property • Preserve all FDs • Small redundancy (not as small as BCNF) • Update/Delete Anomalies some cases • Only one split to two or more tables

- Go for BCNF if we can find a BCNF decomposition that preserves all FDs

If such a decomposition cannot be found

- Go for BCNF if preserving all FDs is not important
- Go for 3NF otherwise

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-

Information
Course: CS2102 Database System
Type: Cheat Sheet
Date: May 22, 2025
Author: QIU JINHANG
Link: https://github.com/jhqiu21/Notes

-*-*-*-*-*-*- PLEASE DELETE THIS PAGE! -*-*-*-*-*-*-