Ng Xuan Jun

Relational Algebra:

Unary Operators

Selection, $\sigma_c(R)$: Selects all tuples from a relation R (i.e. rows from a table) that satisfy the selection condition c

Projection, $\pi_l(R)$: Projects all the attributes of a relation specified in list l

Renaming, $\rho_l(R)$: Renames the attributes of a relation R. Formats to write l are as follows:

- $l = (B_1, B_2, \cdots, B_n)$
- $l = (B_i \leftarrow A_i, \cdots, B_k \leftarrow A_k)$

Set Operations:

Union $R \cup S$: Returns a relation with all tuples that are in both R or S

Intersection $R \cap S$: Returns a relation with all tuples that are in both R and S

Set Difference $R - S$: Returns a relation with all tuples that are in R but not in S

Note that set operations need to be **Union-Compatible**

- R and S must have the same number of attributes
- The corresponding attributes have the same or compatible domains
- R and S do not have to use the same attribute names

Binary Operators:

Cross Product (Cartesian Product) $R \times S$: Combines two relations R and S by forming all pairs of tuples from the two relations

- Commutative (With Projection) and Associative
- Suppose $R(A, B, C), and\ S(X, Y)$
- $R \times S$ returns a relation with schema (A, B, C, X, Y) defined as: $R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$

Inner Joins: Only the tuples that satisfy matching criteria are included in the final result. Commutative (With Projection) and Associative

θ −Join $R \bowtie_\theta S$: We can have any kind of selection conditions

- $R \bowtie_\theta S = \sigma_\theta(R \times S)$
- Contains all attributes of both relations

Equi-Join $R \bowtie_c S$: This is a special case of the θ −Join where it is defined over the equality operator (=) only

- Contains all attributes of both relations

Natural-Join $R \bowtie S$: Same as equi join but it is performed over all attributes that R and S have in common (The matching condition does not need to be specified)

- The output relations contain the common attributes of R and S only once
- If no shared attribute, becomes Cross Product

Outer Join: Tuples that do not satisfy the matching criteria are included in the final result

- Performs inner join first $M = R \bowtie_\theta S$
- To M, add dangling tuples from
  - R in the case of left outer join ⟕
  - S in the case of right outer join ⟖
  - R and S in the case of a full outer join ⟗
- "Pad" missing values of dangling tuples with null

Dangling Tuples: Tuples which do not satisfying the matching criteria and don't match with the tuples of the other relation

$dangle(R \bowtie_\theta S)$: Set of dangling tuples in R w.r.t $R \bowtie_\theta S$

$null(R)$: n-component tuple of null values where n is the number of attributes of R

Left Outer Join $R ⟕ S$: $R \bowtie_\theta S \cup (dangle(R \bowtie_\theta S) \times null(S))$

Right Outer Join $R ⟖ S$: $R \bowtie_\theta S \cup (null(R) \times dangle(S \bowtie_\theta R))$

Full Outer Join $R ⟗ S$: $R \bowtie_\theta S \cup \big((dangle(R \bowtie_\theta S) \times null(S)) \cup (null(R) \times dangle(S \bowtie_\theta R))\big)$

Null Values:

Acceptance condition: WHERE clause (accept on TRUE)

For WHERE clause, if we have NULL values, unless they evaluate to TRUE, we will not accept them

- Note if we want to find tuples with NULL or not-NULL attribute values, we need to use the NULL and NOT NULL keywords.
- By using x <> NULL or x = NULL, it will return us an unknown and therefore, it doesn't accept the rows and therefore, not giving the desired results.

Rejection condition: CHECK constraint (reject on FALSE)

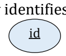
For CHECK constraints, if we have NULL values, unless they evaluate to FALSE, we will not reject them

- Note that if we check x <> NULL then it will be evaluated as unknown and therefore, it is not FALSE and it wont be rejected. Therefore, when we check for NULL values, use IS NULL or IS NOT NULL so that we can correctly get a TRUE or FALSE value

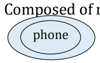
ER Model:

Attributes: In Ovals

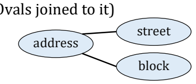
1. **Key Attributes**: Uniquely identifies each entity (Underlined)



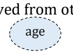
2. **Composite Attributes**: Composed of multiple attributes (Double Oval)



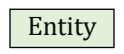
3. **Multivalued Attribute**: May consist of more than one value for a given entity (Other Ovals joined to it)



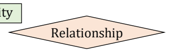
4. **Derived Attribute**: Derived from other attributes (Dashed Oval)



Entities: In Rectangles



Relationships: In Diamonds



Cardinality Constraints: Tells us the **upper limit** of which an entity can participate in a relationship

Many: Denoted by a solid line

One: Denoted by an arrow

Participation Constraint: Tells us the **lower limit** of which an entity can participate in a relationship

Partial Participation: Denoted by single solid line

Total Participation: Denoted by double solid line

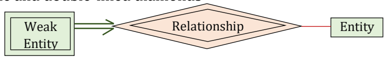
Weak Entity Set: It is an entity set that does not have its own key

- Weak entity can only be uniquely identified by the primary key of its owner entity (A needs entity B to identify each of its entity).

Partial Key: Set of attributes that uniquely identifies a weak entity given an owner entity

- Only if the owner entity exists then the weak entity set exist

Dependency Constraint: For Weak Entity Sets. Indicated by double-lined rectangle and double-lined diamonds

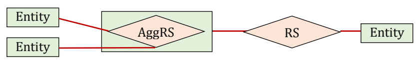


ISA Hierarchy:

Overlapping Constraint: Can a superclass entity belong to multiple subclass (Similar to Cardinality Constraint)

Covering Constraint: Does a superclass entity have to belong to a subclass (Similar to Participation Constraint

Aggregation: Abstraction and treats relationships as entities. Just need to draw a rectangle around the relationship and connect it to a relationship like a normal entity. Note that the connection should be outside of the outer rectangle



SQL:

Create Table: `CREATE TABLE <table>(<attr><type><constraints>…)`

Inserting Data: `INSERT INTO <table>(attrs) VALUES (…)`

DELETE Data: `DELETE FROM <table> WHERE <condition>`

Updating Data: `UPDATE <table> SET <old = new> WHERE <condition>`

Comparison Predicates: `IS NULL, IS DISTINCT FROM`

Type of Constraints: `UNIQUE(…) , NOT NULL(…) , PRIMARY KEY(…),`

Foreign Key Constraint: `FOREIGN KEY (attr) REFERENCES <other_table>(attr) ON DELETE <action> ON UPDATE <action>`
`<action>: NO ACTION, RESTRICT, CASCADE, SET DEFAULT, SET NULL`

CHECK constraint: `CONSTRAINT <name> CHECK <condition>`

- When we have a constraint of $p \Rightarrow q$, convert into $\neg p \vee q$

Deferrable Constraints: `CONSTRAINT <constraint_name> DEFERRABLE INITIALLY (DEFERRED/IMMEDIATE)`

Modifying Schema: `ALTER TABLE <table_name> ALTER COLUMN / ADD COLUMN/ DROP COLUMN/ADD CONSTRAINT/DROP CONSTRAINT`

Dropping Tables: `DROP TABLE (IF EXISTS) <table_name> (CASCADE)`

Conceptual Evaluation of Queries

1. `FROM` – Compute cross product of all tables in FROM clause
2. `WHERE` – Filter tuples that evaluates to true on the WHERE condition(s) (acceptance condition)
3. `GROUP BY` – Partition table into groups wrt to the grouping attribute(s)
4. `HAVING` – Filter groups that evaluates to TRUE on the HAVING condition(s) (acceptance condition)
5. `SELECT` – Remove all attributes not specified in SELECT clause
6. `ORDER BY` – Sort tables based on specified attribute(s)
7. `LIMIT/OFFSET` – Filter tuples based on their order in the table

Expressions in the SELECT Clause:

`DISTINCT`: Removes duplicates

`A AS B`: Renames A to be B

`'abc' || B`: String Concatenation

`round(A)`: rounds A to the nearest integer value

Set Operations:

`UNION, INTERSECT, EXCEPT` (Removes Duplicate Records)

`UNION ALL, INTERSECT ALL, EXCEPT ALL` (Keeps Duplicate Records)

Join Operations:

Inner Join: `R INNER JOIN S ON R.A = S.A`

Left Join: `R LEFT JOIN S ON R.A = S.A`

Right Join: `R RIGHT JOIN S ON R.A = S.A`

Cartesian Product: `FROM R, S`

Subqueries:

`IN` Subquery: `<expr> IN <subquery>, <expr> NOT IN <subquery>`

- Subquery must return exactly one column. Returns **TRUE** if <expr> matches with any subquery row
- IN can be replaced with inner joins and NOT IN can be replaced with outer joins.

`ANY` subquery: `<expr> op ANY <subquery>`

- Subquery must return exactly one column. Returns **TRUE** if comparison evaluates to TRUE for **at least one** subquery row.

`ALL` subquery: `<expr> op ALL <subquery>`

- Subquery must return exactly one column. Returns **TRUE** if comparison evaluates to TRUE for **all** subquery row.

`EXISTS` subquery: `EXISTS<subquery>, NOT EXISTS <subquery>`

- **EXISTS** returns **TRUE** if the subquery returns at least 1 tuple
- **NOT EXISTS** returns **TRUE** if the subquery returns no tuples
- (NOT) EXISTS subqueries are generally always correlated and uncorrelated ones are usually wrong or unnecessary

Aggregate Functions

Signatures

1. `MIN(), MAX()` defined for all data types; return data type same as input data type. **For NULL or Empty: RETURNS NULL**
2. `SUM(), AVG()` defined for all numeric data types; SUM(INTEGER) → BIGINT, SUM(REAL) → REAL. **For NULL or Empty: RETURNS NULL**
3. `COUNT()`, defined for all data types; COUNT(…) → BIGINT. **For NULL or Empty: COUNT(A) FROM R = 0, COUNT(*) FROM R =** $\begin{cases} 0, \text{for Empty} \\ n, \text{for NULL} \end{cases}$

`GROUP BY` Clause – Restrictions to SELECT Clause

If column A_i of table R appears in the SELECT Clause, we need one of the following things to hold:

1. A_i appears in the GROUP BY clause
2. A_i appears as input of an aggregation function in the SELECT clause
3. Primary key of R appears in the GROUP BY clause

`GROUP BY` Clause – Restrictions to HAVING Clause

Note that HAVING clause cannot be used without a GROUP BY clause

If column A_i of table R appears in the HAVING Clause, we need one of the following things to hold

1. A_i appears in the GROUP BY clause
2. A_i appears as input of an aggregation function in the HAVING clause
3. The primary key of the R appears in the GROUP BY clause

CASE – Conditional Expressions

We can use it in the SELECT statement to create certain columns that have varying values depending on certain conditions

Version 1 (When we have differing kind of conditions)	Version 2 (When we have similar conditions but the values is different)

`CASE`
`WHEN condition1 THEN result1`
`WHEN condition2 THEN result2`
`…`
`WHEN conditionn THEN resultn`
`ELSE result0`
`END`
	`CASE expression`
	`WHEN value1 THEN result1`
	`WHEN value2 THEN result2`
	`…`
	`WHEN valuen THEN resultn`
	`ELSE result0`
	`END`

COALESCE – Conditional Expressions for NULL Values

`COALESCE(value1, value2, value3, …)`

Returns the first non-NULL value in the list of input arguments

Returns **NULL** if all values in the list of input arguments are NULL

Can be used normally when we want to give some sort of default value for a table if the column value is NULL.

- SELECT COALESCE(type, 'other') – If the type column is NULL, then the value given will be 'other'

NULLIF – Conditional Expressions for NULL Values

`NULLIF(value1, value2)`

Returns NULL if value1 = value2; otherwise it returns value1

Useful when we want to convert some kind of special values into NULL values

- E.g. (0, "") into NULL values which could be better if we want to do some sort of aggregation calculations

Common Table Expressions (CTE)

General syntax

1. Each C_i is the name of a temporary table defined by query Q_i
2. Each C_i can reference any other C_j that has been declared before C_i
3. SQL statement S can reference any possible subset of all C_i

```
WITH
C1 AS (Q1), C2 AS (Q2), ..., Cn AS (Qn)
SQL statement S;
```

Views

Permanently named query (which is kind of like a virtual relation)

Can be used like some normal tables (with some restrictions)

The result of a query is not permanently stored (query is executed each time the view is used)

```
CREATE VIEW <name> AS
SELECT …
FROM …
…;
```

Note:

When we INSERT, UPDATE, DELETE from the views, it should not cause any major changes to our original table

Universal Quantification:

SQL doesn't really support universal quantification, but it supports existential quantification (EXSITS)

$$\forall x \Rightarrow \neg \exists x$$

We can convert our universal quantifications into existential quantification using the above relation that we have from C1231

For example:

"all users who visited all countries" → "there does not exists a country that the user has not visited"

Recursive Queries

For recursive queries where we need to recursively compute something with our previous query, we can make use of CTES

```
WITH RECURSIVE cte_name AS (
Q1
UNION [ALL]
Q2(cte_name)
)
SELECT * FROM cte_name;
```

Where Q_1 is our base case

Q_2 is the recursive term that will reference to the base case with the cte_name and recursively do the computation. Note that we can state the stopping condition in the WHERE clause

UNION ALL can be used if we do not want to remove duplicates, use UNION only if we want to keep distinct values only

Ng Xuan Jun

Functions / Procedures

Used when we have a return type. If there is no return type, just use a Procedure

Functions	Procedures
Returns a value Invoke using `SELECT`	Does not return a value Invoke using `CALL`

General Syntax:
`CREATE OR REPLACE FUNCTION`
`<name>`
`(<param><type>,`
`<param><type> … )`
`RETURNS <type> AS $$`
`BEGIN`
`<code>`
`END;`
`$$ LANGUAGE <language>`
	General Syntax:
	`CREATE OR REPLACE PROCEDURE`
	`<name>`
	`(<param><type>,`
	`<param><type> … ) AS $$`
	`BEGIN`
	`<code>`
	`END;`
	`$$ LANGUAGE <language>`

<language> can be sql, plpgsql

Return Types (For Functions):

RETURNS	<type>
Single Existing Tuple	<table_name>
Set of existing tuple	SET OF <table_name>
Single new tuple	RECORD
Set of new tuple	SET OF RECORD/ TABLE(…)
Triggers	TRIGGER

Control Structures:

Type	Syntax
Variables	**Declaration**: `DECLARE [<var_name> <type>]` **Assignment**: `<var_name> := <expression>`

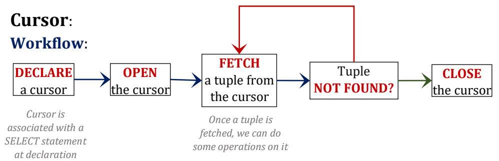
Selection	`IF <condition> THEN <statement>`
	`ELSIF <condition> THEN <statement>`
	`ELSE <statement>`
	`END IF;`

Repetition	`LOOP`
	`EXIT <label> WHEN <condition>;`
	`<statement>`
	`END LOOP;`
	`WHILE<condition> LOOP`
	`<statement>`
	`END LOOP;`
	`FOR <var_name> IN <lower> .. <upper>`
	`LOOP`
	`<statement>`
	`END LOOP;`

Blocks	`BEGIN`
	`...`
	`END`

Cursor:

Workflow:



General Syntax:

```
DECLARE
  curs CURSOR FOR (SELECT … );
  r RECORD;
BEGIN
  OPEN curs;
  LOOP
  FETCH curs INTO r; EXIT WHEN NOT FOUND;
  -- do operations
  RETURN NEXT; -- inputs the current values into the resultant table
  END LOOP; CLOSE curs; END;
```

Movements:

Starts from before the first row and it moves according to the movement stated

Type	Call
Next row	`FETCH curs INTO r`
From previous row	`FETCH PRIOR FROM curs INTO r`
From top row	`FETCH FIRST FROM curs INTO r`
From last row	`FETCH LAST FROM curs INTO r`
From nth row	`FETCH ABSOLUTE n FROM curs INTO r`

Triggers

Syntax

Trigger Function	Trigger
Needs to be created before Trigger	Calls on Trigger Function

`CREATE OR REPLACE FUNCTION`
`<trigger_name>(<param><type>,`
`<param><type> … )`
`RETURNS TRIGGER AS $$`
`BEGIN`
`<code>`
`END;`
`$$ LANGUAGE <language>`
	`CREATE Trigger <name>`
	`<trigger_timing>`
	`<trigger_event>`
	`ON <table>`
	`FOR EACH <trigger_granularity>`
	`[WHEN <condition>]`
	`EXECUTE FUNCTION`
	`<trigger_name>()`

Keywords

Type	Keywords
Trigger Events <`trigger_event`> Can check for multiple events using OR	`INSERT, DELETE, UPDATE`
Trigger Timing <`trigger_timing`>	`BEFORE, AFTER`

Transition Variables	`OLD` – Only for `DELETE, UPDATE`
	`NEW` – Only for `INSERT, UPDATE`
	`TG_OP` – Operation name

Trigger Granularities `<trigger_granularity>`	`FOR EACH ROW` `FOR EACH STATEMENT`
Runs the trigger only when the condition is met **Limitations**: No `SELECT` in `WHEN()` No `OLD` in `WHEN()` for `INSERT` No `NEW` in `WHEN()` for `DELETE` No `WHEN()` for `INSTEAD OF`	`WHEN (<condition>)`

Return Values (Row-Level)

Events + Timing	NULL Tuple	Non-NULL Tuple
BEFORE INSERT/UPDATE/DELETE	Operation cancelled	Operation as per normal
AFTER INSERT/UPDATE/DELETE	**NO EFFECT** (RAISE EXCEPTION if need to undo)	

Return Values (Statement-Level)

Does not matter, RAISE Exception if need to undo

Deferred Triggers (Only Row-Level, AFTER): Triggers that are checked only at the end of the transaction instead of each statement

`INITIALLY DEFERRED` – Means the trigger is Deferred by default

`INITIALLY IMMEDIATE` – Means the trigger is NOT Deferred by default

Syntax:

```
CREATE CONSTRAINT TRIGGER <trigger_name>
<trigger_timing><trigger_event> ON <trigger table>
[DEFERRABLE INITIALLY [DEFERRED | IMMEDIATE]]
FOR EACH <trigger_granularity>
[WHEN <trigger_condition>]
EXECUTE FUNCTION <trigger_function_name> ();
```

INSTEAD OF Trigger (Only Row-Level): Will execute the trigger function instead of the actual operation. Used only for views.

Functional Dependencies:

For a Functional Dependency $X \to Y$. If they have the same value for X, they must have the same value for Y. ($\emptyset \to A$ means that A is a constant value)

Trivial: RHS is subset of LHS. ($\emptyset \subset \emptyset$ so $\emptyset \to \emptyset$ is trivial)

Non-Trivial: RHS is not subset of LHS

Completely Non-Trivial: RHS is nonempty and no attributes on RHS appear on the LHS. ($A \not\subset \emptyset$ so $\emptyset \to A$ is completely non - trivial)

Let R be a relation, $S \subset R$ be a set of attributes. For $S \to R$

Computing Projected FDs:

1. For each of the fragments, look at all possible subsets of attributes. Check for the "More" part of an attribute closure (can be "All") also.
2. The "More" will be the projected FD.

E.g. $R = \{A, B, C, D\}, \{A\}^+ = \{AB\}$ then we can say that $\{A\} \to \{B\}$

Keys

Superkey: S should imply the whole relation R.

Candidate Key: S is a minimal superkey

Primary Key: Candidate key that the designer chooses

Prime Attribute: An attribute that **appears in some candidate key** of R with Σ (otherwise it is called non-prime attribute)

Tricks:

1. Check through the **Attribute Closures.** Start from the smallest attribute set and once we reach an attribute set that implies the relation, it is a candidate key and we don't have to check any of its supersets since those will Superkeys.
2. If an attribute doesn't appear on RHS of any FD, then it must be part of every candidate key

Closures:

The closure of Σ, denoted at Σ^+ is the set of all functional dependencies **logically entailed** by the functional dependencies in Σ.

Equivalence of Functional Dependencies:

Two sets of functional dependencies Σ and Σ' are equivalent if and only if they have the **same closure:** $\Sigma \equiv \Sigma', \Sigma^+ \equiv \Sigma'^+$

Cover: Σ' is a cover of Σ (and Σ is a cover of Σ') if and only if $\Sigma \equiv \Sigma'$. They are equivalent.

Attribute Closures: The closure of a set of attributes $S \subset R$ (denoted S^+) is the **set of all attributes that are functionally dependent** on S.

Minimal Set of Functional Dependencies:

1. **RHS** of every functional dependency is **minimal**. RHS should only be a singleton
2. **LHS** of every functional dependency is **minimal**. There should be no functional dependency where they have the same RHS but one of the LHS is a subset of the other.
3. The **set itself should be minimal**. None of the functional dependencies in Σ can be derived from other functional dependencies in Σ.

Minimal Cover:

A minimal cover of a set of functional dependencies is a set of functional dependencies Σ' that is both minimal and equivalent to Σ

Every set of functional dependencies has a minimal cover (**Not Unique**)

Algorithm for Minimal Cover:

1. Simply the RHS to singleton for every functional dependency to get Σ'
2. Simplify the LHS to ensure step 2 of Minimal Set to get Σ''
3. Ensure that no functional dependency can be derived from other functional dependencies to get Σ'''

Compact:

There are no different functional dependencies with the same LHS. If they have the same LHS, just combine the RHS together.

Compact Cover:

Set of functional dependencies that is both **compact** and **equivalent** to Σ.

Every set of functional dependencies has a compact cover

Compact Minimal Cover:

Set of functional dependencies that is both **compact, minimal and equivalent** to Σ

Note that a Compact Minimal Cover is **not a Minimal Cover** since the RHS is not singleton.

Every set of functional dependencies has a compact minimal cover.

Algorithm for Compact Minimal Cover:

1. Do Steps 1-3 of a Minimal Cover
2. If any functional dependencies have the same LHS, combine their RHS together

Armstrong Axioms:

Let R be a set of attributes

Reflexivity: $\forall X \subset R, \forall Y \subset R, \ ((Y \subset X) \Rightarrow (X \to Y))$

Augmentation: $\forall X \subset R, \forall Y \subset R, \forall Z \subset R, \ ((X \to Y) \Rightarrow (X \cup Z \to Y \cup Z))$

Transitivity: $\forall X \subset R, \forall Y \subset R, \forall Z \subset R, \ ((X \to Y \land Y \to Z) \Rightarrow (X \to Z))$

Types of Anomalies:

Solution: Storing it in a separate table could allow us to enforce such a constraint

1. **Redundant Storage :** There are repeats of column entries that may not be required. Waste of space
2. **Update Anomalies :** We may not be able to enforce certain functional dependencies and updating will cause us to break it
3. **Deletion Anomalies :** If we remove a certain entry, some information could be lost forever.
4. **Insertion Anomalies:** We may not be able to insert certain entries since we will need full information of everything.

Lossless-Join:

A binary decomposition of R into R_1 and R_2 is lossless-join if $R = R_1 \cup R_2$ and $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$. (*The intersecting attributes should the primary key of one of the resultant fragments*)

A decomposition is lossless-join if there exists a sequence of binary lossless-join decomposition that generates that decomposition.

Tricks:

1. Compute the attribute closures.
2. If there are only 2 fragments, check if the intersecting terms is a primary key of either of the fragments, then it is lossless-join
3. If there are 3 or more fragments, try to check either top down or bottom up if the binary decomposition is lossless-join as per (2).

BCNF:

A relation R with a set of functional dependencies Σ is in **BCNF** if and only if every functional dependency $X \to \{A\} \in \Sigma^+$:

1. $X \to \{A\}$ is **trivial** or
2. X is a **superkey**

BCNF Decomposition Algorithm:

- Guaranteed lossless-join decomposition in BCNF
- May not be dependency preserving

1. Find the functional dependency $X \to Y \in \Sigma$ that violates BCNF
2. Decompose R into the relations R_1 and R_2 of the following form. Find the attributes that are in X's closure and put them in one relation and the other one is just the remaining with X:
$$R_1 = X^+, \quad R_2 = (R - X^+) \cup X$$
3. Check whether R_1 and R_2 with the respective projected functional dependencies Σ_1 and Σ_2 are in BCNF. If any of them are not, continue Step 1 and 2 on that relation

Tricks:

1. Calculate the attribute closures.
2. For each of the fragments, check for all the subsets of the attributes in each fragment. If there are any fragments with the "More but not All" property, then BCNF is violated.

More but not All – Attribute closure implies more attributes than the LHS but it is not the entire relation. We can also use this to identify possible FDs that have this feature because the "More" part are the relations that it implies.

VERY USEFUL FOR COMPUTING HIDDEN DEPENDENCIES

E.g. $R(A, B, C), \{A\}^+ = \{A, B\}$ means that we have $A \to B$ since it is the more part, if it is $\{A\}^+ = \{A, B, C\}$ means that we have $A \to BC$

3NF:

A relation R with a set of functional dependencies Σ is in **3NF** if and only if every functional dependency $X \to \{A\} \in \Sigma^+$:

1. $X \to \{A\}$ is **trivial** or
2. X is a **superkey or**
3. A is a prime attribute

Note: If it **satisfies** BCNF, it also **satisfies** 3NF. If it **violates** 3NF, it also **violates** BCNF

Synthesis (Bernstein Algorithm):

Guaranteed to have a lossless, dependency preserving decomposition in 3NF.

Use a compact minimal cover for this

1. Construct a compact minimal cover for the set of functional dependencies Σ. (*Using the algorithm for Compact Minimal Cover*)
2. For each functional dependency $X \to Y$ in the minimal cover create a relation $R_i = X \cup Y$ unless it already exists or is subsumed by another relation
3. **(Remember to Check)** If none of the created relations contain one of the keys, pick a candidate key and create a relation with

Tricks:

1. Calculate the attribute closures
2. For each of the fragments, check for all the subsets of the attributes in each fragment. If there are any fragments with the "More but not All" property, check if the "More" part is a prime attribute (part of any candidate key)

E.g. $R(A, B, C, D), \{A\}^+ = \{A, B\}, \{B, C\} \to \{A, B, C, D\}$. A has more but not all, B is the more part but it is part of the candidate key $\{B, C\}$ so it is prime attribute and no violations.

Dependency-Preserving:

Let Σ be the original set of FDs, and suppose we decompose R into R_1 and R_2 with the projected FDs of Σ_1 and Σ_2. The decomposition is dependency preserving if (i.e. we can derive the original FDs from the decomposed one, vice-versa): $\Sigma = \Sigma_1 \cup \Sigma_2$

Tricks:

1. Compute all the projected FDs of the decomposed relations.
2. Union all the projected FDs together.
3. Check if the closure of the original FDs are still the same.

E.g. Union of Decomposed $= \{\{A\} \to \{B, C\}, \{C\} \to \{D\}\}$, Original $\{\{A\} \to \{B, C, D\}, \{C\} \to \{D\}\}$. Check that using the new FD, the closure of $\{A\}, \{C\}$ still has the "More" part as $\{B, C, D\}$ and $\{D\}$ respectively.