CS2102 Cheatsheet AY22/23 Sem 1

by Richard Willie

Relational Model

Structure of Relational Databases

- A **relational database** consists of a collection of **tables**, each of which is assigned a unique name.
- A **tuple** refers to a row of a table.
  - A row in a table represents a *relationship* among a set of values.
- An **attribute** refers to a column of a table.
  - For each attribute, there is a set of permitted values, called the **domain** of that attribute.
  - The domains of all attributes must be **atomic**. A domain is atomic if elements of the domain are considered indivisible.
  - Formalism:

$$dom(A_i) = \text{domain of attribute } A_i$$
$$\text{For each value } v \text{ of } A_i,\ v \in dom(A_i) \text{ or } v = null$$

  - The **null value** is a special value that signifies that the value is unknown or does not exist.
- A **relation** refers to a table.
  - Formalism:

$$R(A_1, A_2, \ldots, A_n) = \text{A } \textbf{relation schema} \text{ with name } R \text{ and } n \text{ attributes}$$
$$\text{Each instance of schema } R \subseteq \{(a_1, a_2, \ldots, a_n) : a_i \in dom(A_i) \cup \{null\}\}$$

Integrity Constraints

- An **integrity constraint** is a condition that restricts what constitutes valid data.
- Three main structural integrity constraints:
  - Domain constraints
  - Key constraints
  - Foreign key constraints

Key Constraints

- We must have a way to specify how tuples within a given relation are distinguished. This is expressed in terms of their attributes.
- A **superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.
  - A superkey may contain extraneous attributes.
  - If K is a superkey, then so is any superset of K.
- A **candidate key** is a minimal superkey for which no proper subset is a superkey.
  - It is possible that several distinct sets of attributes could serve as a candidate key.
- The term **primary key** denote a candidate key that is chosen as the principle means of identifying tuples within a relation.
- A key (whether primary, candidate, or super) is a property of the entire relation, rather than the individual tuples.

Foreign Key Constraints

- A **foreign key constraint** from attribute(s) A of relation R_1 to the primary-key B of relation R_2 states that on any database instance, the value of A for each tuple in R_1 must also be the value of B for some tuple in R_2.
- Attribute set A is called a **foreign key** from R_1, referencing R_2.
- The relation R_1 is also called the **referencing relation** of the foreign-key constraint, and R_2 is called the **referenced relation**.
- Each foreign key in a referencing relation must either:
  - appear as a primary key in the referenced relation, or
  - be a *null* value.

Relational Algebra

Unary Operators

Selection σ_c

- $\sigma_c(R)$ selects all tuples from a relation R that satisfy the *selection condition* c.
- Formalism:

$$\forall t \in R,\ t \in \sigma_c(R) \iff c \text{ evaluates to } \textbf{true} \text{ on } t$$

- Rules of handling *null* values:
  - The result of a comparison operation with *null* is ***unknown***.
  - The result of an arithmetic operation with *null* is ***null***.
- Equivalent SQL:

```
SELECT * FROM R WHERE c;
```

Projection π_l

- $\pi_l(R)$ projects all the attributes (columns) of a relation specified in list l.
- Note that duplicate tuples are removed from the output relation.
- Equivalent SQL:

```
SELECT DISTINCT col_1, col_2, ... FROM R;
```

Renaming ρ_l

- $\rho_l(R)$ renames the attributes of a relation R (with schema $R(A_1, A_2, \ldots, A_n)$).
- Two popular formats for l:
  - $l = (B_1, B_2, \ldots, B_n)$
  - $l = B_1 \leftarrow A_1, \ldots, B_n \leftarrow A_n$

Set Operators

- Three set operators:
  - **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
- Two relations R and S are **union-compatible** if
  - R and S have the same number of attributes, and
  - the corresponding attributes have the same or compatible domains, but
  - R and S do not have to use the same attribute names.

Cross Product $\times$

- The **cross product** combines two relations R and S by forming all pairs of tuples from the two relations.
- More formally, given two relations $R(A, B, C)$ and $S(X, Y)$:
  - $R \times S$ returns a relation with schema (A, B, C, X, Y)
  - $R \times S = \{(a, b, c, x, y) : (a, b, c) \in R, (x, y) \in S\}$
- Equivalent SQL:

```sql
SELECT * FROM R, S;
```

Join Operators

Inner Joins

- Three types:
  - θ-join $\bowtie_\theta$
  - Equi join $\bowtie$
  - Natural join $\bowtie$
- The **θ-join** of two relations R and S is defined as

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

  where θ is an arbitrary comparison operators.
- **Equi join** is a special case of θ-join where θ is the equality operator $(=)$.
- **Natural join** is equivalent to equi join, but
  - the join is performed over all attributes that R and S have in common;
  - the output relations contains the common attributes of R and S only once.
- More formally, the natural join of two relation R and S is defined as

$$R \bowtie S = \pi_l(R \bowtie_c \rho_{b_i \leftarrow a_i, \ldots, b_k \leftarrow a_k}(S))$$

  where:
  - $A = \{a_i, \ldots, a_k\}$ is the set of attributes that R and S have in common
  - $c = ((a_i = b_i) \wedge \cdots \wedge (a_k = b_k))$
  - $l =$ (list of all attributes of R) + (list of all attributes of S that are not in A)
- Equivalent SQL:

```sql
-- theta-join
SELECT * FROM R JOIN S ON condition;

-- natural join
SELECT * FROM R NATURAL JOIN S;
```

Outer Joins

- Motivation:
  - Inner joins eliminate all tuples that do not satisfy matching criteria (i.e., attribute selection).
  - Sometimes the tuples in R or S that do not match with tuples in the other relation are of interest, also known as **dangling tuples**.
- Auxiliary definitions:
  - $dangle(R \bowtie_\theta S)$ = 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 in R
- Formalism:
  - **Left outer join**:

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

  - **Right outer join**:

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

  - **Full outer join**:

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

- Natural outer joins is analogous to natural inner joins.
- Equivalent SQL:

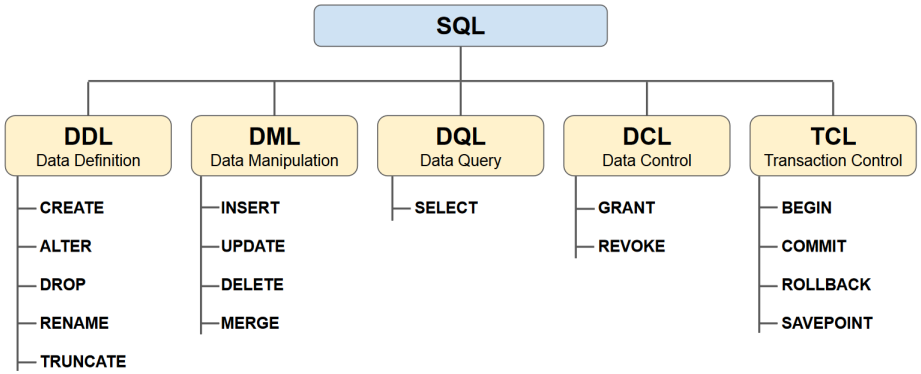
```sql
-- left outer join
SELECT * FROM R LEFT OUTER JOIN S ON condition;

-- right outer join
SELECT * FROM R RIGHT OUTER JOIN S ON condition;

-- full outer join
SELECT * FROM R FULL OUTER JOIN S ON condition;
```

Equivalence Transformation Rules

1. $\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma\theta_2(E)) = \sigma_{\theta_2}(\sigma\theta_1(E))$
2. $\pi_{L_1}(\pi_{L_2}(\ldots \pi_{L_n}(E)\ldots)) = \pi_{L_1}(E)$
3. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
4. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
5. $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
6. $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
7. $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
8. $\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_0}(E_1) \bowtie_\theta E_2$
9. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie_\theta \sigma_{\theta_2}(E_2)$
10. $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1}(E_1) \bowtie_\theta \pi_{L_2}(E_2)$
11. $\pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \pi_{L_1 \cup L_2}(\pi_{L_1 \cup L_3}(E_1) \bowtie_\theta \pi_{L_2 \cup L_4}(E_2))$
12. $\sigma_p(E_1 \cup E_2) = \sigma_p(E_1) \cup E_2 = \sigma_p(E_1) \cup \sigma_p(E_2)$
13. $\sigma_p(E_1 \cap E_2) = \sigma_p(E_1) \cap E_2 = \sigma_p(E_1) \cap \sigma_p(E_2)$
14. $\sigma_p(E_1 - E_2) = \sigma_p(E_1) - E_2 = \sigma_p(E_1) - \sigma_p(E_2)$
15. $\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$

SQL

Types of SQL Commands/Statements



DDL (Data Definition Language)

Creating Table

- Example:

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

Altering Table

- Example:

```
-- Change data type
ALTER TABLE Projects ALTER COLUMN name TYPE VARCHAR(200);

-- Set default value
ALTER TABLE Projects ALTER COLUMN start_year SET DEFAULT 2021;

-- Drop default value
ALTER TABLE Projects ALTER COLUMN start_year DROP DEFAULT;

-- Add new column with a default value
ALTER TABLE Projects ADD COLUMN budget NUMERIC DEFAULT 0.0;

-- Drop column from table
ALTER TABLE Projects DROP COLUMN budget;

-- Add foreign key constraint
ALTER TABLE Teams ADD CONSTRAINT eid_fkey FOREIGN KEY (eid) REFERENCES
    Employees (id);

-- Drop foreign key constraint
ALTER TABLE Teams DROP CONSTRAINT eid_fkey;
```

Dropping Table

- Example:

```
-- Check if table exists before delete to avoid throwing an error
DROP TABLE IF EXISTS Projects;

-- Throw an error if foreign key constraint exists, otherwise delete
DROP TABLE Projects;

-- Delete Projects and foreign key constraint
DROP TABLE Projects CASCADE;
```

Data Integrity Constraints in SQL

Not-Null Constraints

- Example:

```
-- unnamed constraint (named assigned by DBMS)
CREATE TABLE Employees (
    id      INTEGER NOT NULL,
    name    VARCHAR(50) NOT NULL,
    ...
);

-- named constraint (easier bookkeeping)
CREATE TABLE Employees (
    id      INTEGER CONSTRAINT nn_id NOT NULL,
    name    VARCHAR(50) CONSTRAINT nn_name NOT NULL,
    ...
);
```

- Not-null constraint violations:
  - There exists a tuple $t \in$ Employees where `t.id IS NOT NULL` evaluates to ***false***.
  - There exists a tuple $t \in$ Employees where `t.name IS NOT NULL` evaluates to ***false***.

Unique Constraints

- Example:

```
-- unnamed column constraint
CREATE TABLE Employees (
    id      INTEGER UNIQUE,
    ...
);

-- unnamed table constraint
CREATE TABLE Teams (
    id      INTEGER,
    name    VARCHAR(100),
    ...
```

```
    UNIQUE (id, name)
);
```

- Unique constraints for more than one attribute/column can only be specified using table constraints.
- Unique constraint violations:
  - For any two tuples $t_i, t_k \in$ Teams, (`$t_i$.id <> $t_k$.id`) or (`$t_i$.name <> $t_k$.name`) evaluates to ***false***.

Primary Key Constraints

- Example:

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    ...
);
```

- `PRIMARY KEY` has the same effect as `UNIQUE NOT NULL`. However, only one `PRIMARY KEY` constraint can be defined, while there may be multiple `UNIQUE NOT NULL` constraints.

Foreign Key Constraints

- Example:

```
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    ...
);

CREATE TABLE Projects (
    name    VARCHAR(50) PRIMARY KEY,
    ...
);

CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(50),
    ...
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id),
    FOREIGN KEY (pname) REFERENCES Projects (name)
);
```

- We can specify actions in case of violation of a foreign key constraint with `ON DELETE/UPDATE <ACTION>`.
- Possible actions for `ON DELETE/UPDATE`:
  - `NO ACTION` – rejects delete/update if it violates constraint (default value)
  - `RESTRICT` – similar to "no action" except that check of constraint cannot be deferred
  - `CASCADE` – propagates delete/update to referencing tuples
  - `SET DEFAULT` – updates foreign keys of referencing tuples to some default value (must be a primary key in the referenced table)
  - `SET NULL` – updates foreign keys of referencing tuples in to *null* (corresponding column must be allowed to contain *null* values)
- Example:

```
CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(50),
    ...
    PRIMARY KEY (eid, pname),
    FOREIGN KEY (eid) REFERENCES Employees (id) ON DELETE NO ACTION ON
        UPDATE CASCADE,
    FOREIGN KEY (pname) REFERENCES Projects (name) ON DELETE SET NULL
        ON UPDATE CASCADE
);
```

Check Constraints

- A general constraint, not a structural integrity constraint.
- Example:

```
-- Constraint for one column
CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(50),
    hours   INTERGER check (hours > 0),
    ...
);

-- Constraint can refer to multiple columns
CREATE TABLE Projects (
    name       VARCHAR(50) PRIMARY KEY,
    start_year INTEGER,
    end_year   INTEGER,
    CONSTRAINT valid_lifetime CHECK (start_year <= end_year)
);

-- Constraint can be arbitrarily complex
CREATE TABLE Teams (
    eid     INTEGER,
    pname   VARCHAR(50),
    hours   INTERGER,
    ...
    CHECK (
        (pname = 'CoreOS' AND hours >= 30)
        OR
        (pname <> 'CoreOS' AND hours > 0)
    )
);
```

Deferrable Constraints

- By default, constraints are checked immediately at the end of SQL statement execution. A violation will cause the statement to be rolled back.
- Constraint checking can be deferred for `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints.
- Options:
  - `NOT DEFERRABLE` – immediately check for constraint, and cannot be changed (default)
  - `DEFERRABLE INITIALLY IMMEDIATE` – immediately check for constraint, but can be changed
  - `DEFERRABLE INITIALLY DEFERRED` – defer check for constraint, but can be changed
- Example:

```sql
CREATE TABLE Employees (
    id      INTEGER PRIMARY KEY,
    name    VARCHAR(50),
    manager INTEGER,
    FOREIGN KEY (manager) REFERENCES Employees (id) DEFERRABLE
        INITIALLY DEFERRED
);

INSERT INTO Employees VALUES (101, 'Sarah', null), (102, 'Judy', 101),
    (103, 'Max', 102);

BEGIN;
DELETE FROM Employees WHERE id = 102; -- Judy got fired, constraint
    violated for Max but not checked
UPDATE Employees SET manager = 101 WHERE id = 103; -- Max gets a new
    manager, constraint re-established
COMMIT;
```

- Benefits:
  - No need to care about order of SQL statements within a transaction.
  - Allows for cyclic foreign key constraints.
  - Performance boost when constraint checks are bottleneck.
- Downsides:
  - Troubleshooting can be more difficult.
  - Data definition no longer unambiguous.
  - Performance penalty when performing queries.

DML (Data Manipulation Language)

Inserting Data

- Example:

```sql
-- Specify all attribute values
INSERT INTO Employees VALUES (101, 'SARAH', 25, 'dev');

-- Specify selected attribute values
INSERT INTO Employees (id, name) VALUES (102, 'Judy'), (103, 'Max');
```

Deleting Data

- Example:

```sql
-- Delete all tuples
DELETE FROM Employees;

-- Delete selected tuples
DELETE FROM Employees
WHERE role = 'dev';
```

Updating Data

- Example:

```sql
UPDATE Employees
SET age = age + 1
WHERE name = 'Sarah';

UPDATE Employees
SET name = UPPER(name),
    role = UPPER(role);
```

ER Model

Participation and Cardinality Constraints

$\rightarrow$	0..1
$-$	0..$*$
$\Rightarrow$	1..1
$=$	1..$*$

Functional Dependencies

Definitions

- An instance r (a table) of a relation schema R satisfies the **functional dependency** σ: $X \rightarrow Y$ with $X \subset R$ and $Y \subset R$, if and only if two tuples of r agree on their X-values, then they agree on their Y-values.
- A functional dependency $X \rightarrow Y$ is **trivial** if and only if $Y \subset X$.
- A funcional dependency is **non-trivial** if and only if $Y \not\subset X$.
- A functional dependency is **completely non-trivial** if and only if $Y \neq \emptyset$ and $Y \cap X = \emptyset$.
- S is a **superkey** of R if and only if $S \rightarrow R$, where R is a relation and $S \subset R$ is a set of attributes of R.
- S is a **candidate** of R is and only if $S \rightarrow R$ and for all $T \subset S$, $T \neq S$, T is not a superkey of R. In other words, S is a minimal superkey.
- A **prime attribute** is an attribute that appears in some candidate key of R with with Σ (a set of functional dependencies on R).

Closure and Equivalence

- The **closure** of Σ, noted Σ^+, is the set of all functional dependencies logically entailed by the functional dependencies in Σ.
- Two sets of functional dependencies Σ and Σ' are **equivalent** if and only if they have the same closure.

$$\Sigma \equiv \Sigma' \iff \Sigma^+ \equiv \Sigma'^+$$

- The **closure** of a set of attributes $S \subset R$, noted S^+, is the set of all attributes that are functionally dependent on S.

$$S^+ = \{A \in R \mid \exists\ (S \to \{A\}) \in \Sigma^+\}$$

- Attribute closure algorithm:

```
input : S, Σ
output: S+
begin
    Ω := Σ;     // unused
    Γ := S;     // closure
    while X → Y ∈ Ω and X ⊂ Γ do
        Ω := Ω − {X → Y};
        Γ := Γ ∪ Y;
    return Γ;
```

Armstrong Axioms

- The following inference rules are **Armstrong Axioms**.
  - **Reflexivity**

$$\forall X \subset R,\ \forall Y \subset R\ \ (Y \subset X) \implies (X \to Y)$$

  - **Augmentation**

$$\forall X \subset R,\ \forall Y \subset R,\ \forall Z \subset R\ \ (X \to Y) \implies (X \cup Z \to Y \cup Z)$$

  - **Transitivity**

$$\forall X \subset R,\ \forall Y \subset R,\ \forall Z \subset R\ \ (X \to Y\ \vee\ Y \to Z) \implies (X \to Z)$$

- Other axioms:
  - **Weak reflexivity** is sound.

$$\forall X \subset R\ \ (X \to \emptyset)$$

  - **Weak augmentation** is sound.

$$\forall X \subset R,\ \forall Y \subset R,\ \forall Z \subset R\ \ (X \to Y) \implies (X \cup Z \to Y)$$

Minimal Cover

- A set of functional dependencies Σ is **minimal** if and only if:
  - The RHS of every functional dependency in Σ is minimal. Namely, every functional dependency is of the form $X \to \{A\}$.
  - The LHS of every functional dependency is minimal. Namely, for every functional dependency in Σ of the form $X \to \{A\}$ there is no functional dependency $Y \to \{A\}$ in Σ^+ such that Y is a proper subset of X.
  - The set itself is minimal. Namely, none of the functional dependency in Σ can be derived from other functional dependencies in Σ.
- A **minimal cover** of a set of functional dependencies Σ is a set of functional dependencies Σ' that is both minimal and equivalent to Σ.
- An algorithm for the computation of minimal cover:
  1. Simplify the RHS of every functional dependency in Σ to get Σ'.
  2. Simplify the LHS of every functional dependency in Σ' to get Σ''.
  3. Simplify the set Σ'' to get Σ'''.
- A set of functional dependencies Σ is **compact** if and only if there is no different functional dependencies with the same left-hand side.
  - $\Sigma = \{\{A\} \to \{B\}, \{A\} \to \{C\}\}$ is not compact.
  - $\Sigma = \{\{A\} \to \{B, C\}$ is compact.
- A **compact cover** of a set of functional dependencies Σ is a set of functional dependencies Σ' that is both compact and equivalent to Σ.
- A **compact minimal cover** (or **canonical cover**) of a set of functional dependencies Σ is a set of functional dependencies Σ' that is both compact, minimal, and equivalent to Σ.
- An algorithm for the computation of minimal cover:
  1. Simplify the RHS of every functional dependency in Σ to get Σ'.
  2. Simplify the LHS of every functional dependency in Σ' to get Σ''.
  3. Simplify the set Σ'' to get Σ'''.
  4. Regroup all the functional dependencies with the same LHS in Σ''' to get Σ''''.

Boyce-Codd Normal Form

Definition

- A relation R with a set of functional dependencies Σ is in **BCNF** if and only if for every functional dependency $X \to \{A\} \in \Sigma^+$, $X \to \{A\}$ is trivial **or** X is a superkey.
  - Note: it is sufficient to look at Σ.

Decomposition

- A **decomposition** of a table R is a set of tables $\{R_1, R_2, \ldots, R_n\}$ such that $R = R_1 \cup \cdots \cup R_n$.
- A **binary decomposition** of a table R is a pair of tables $\{R_1, R_2\}$ such that $R = R_1 \cup R_2$.

Lossless Decomposition

- A decomposition is **lossless-join** if there exists a *sequence* of binary lossless-join decomposition that generates that decomposition.
- A binary decomposition is lossless-join if and only if the full outer natural join of its two fragments (the two tables resulting from the decomposition) equals the initial table (otherwise it is lossy).
- 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 $R_1 \cap R_2 \to R_2$.
  - Note that if $R_1 \cap R_2$ is the primary key of one of the two tables, then it can be a foreign key in the other referencing the primary key.

Projected Functional Dependencies

- A set of **projected functional dependencies** Σ' on R' from R with Σ, where $R' \subset R$, is the set of functional dependencies equivalent to the set of functional dependencies $X \to Y$ in Σ^+ such that $X \subset R'$ and $Y \subset R'$.

Dependency Preserving Decomposition

- A decomposition of R with Σ into $R_1 \dots R_n$ with the respective sets of projected functional dependencies $\Sigma_1 \dots \Sigma_n$ is **dependency preserving** if and only if $\Sigma^+ = (\Sigma_1 \cup \dots \cup \Sigma_n)^+$.

Decomposition Algorithm

- When a relation if not in BCNF, we can pick one of the functional dependencies violating the BCNF definition and use it to decompose the relation into two relations. We continue decomposing until every fragment is in BCNF.
- The decomposition algorithm is guaranteed to find a lossless decomposition in BCNF.
- The decomposition may not be dependency preserving.
- Algorithm:
  1. Let $X \to Y$ be a functional dependency in Σ that violates the BCNF definition. We use it to decompose R into the following two relations:
     - $R_1 = X^+$
     - $R_2 = (R - X^+) \cup X$
  2. Check whether R_1 and R_2 with the respective sets of projected functional dependencies Σ_1 and Σ_2 are in BCNF and continue of decomposition if they are not.

Third Normal Form

Definition

- A relation R with a set of functional dependencies Σ is in **Third Normal Form** (or **3NF**) if and only if for every functional dependency $X \to \{A\} \in \Sigma^+$, $X \to \{A\}$ is trivial **or** X is a superkey or A is a prime attribute.
  - Note: it is sufficient to look at Σ.

Synthesis (Bernstein Algorithm)

- When a relation is not in 3NF, we can synthesise a schema in 3NF from a minimal cover of the set of functional dependencies.
  1. 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.
  2. If none of the created relations contain one of the keys, pick a candidate key and create a relation with that candidate key.
- In order to avoid unnecessary decomposition, it is generally a good idea to use compact minimal cover.
- The algorithm guarantees a lossless, dependency preserving decomposition in 3NF.

Normal Forms

- There are more normal forms that correspond to functional dependencies as well as other integrity constraints:
  - 4NF $\subset$ BCNF $\subset$ 3NF $\subset$ 2NF $\subset$ 1NF
  - 4NF $\neq$ BCNF $\neq$ 3NF $\neq$ 2NF $\neq$ 1NF