CS2102 Helpsheet – AY2223 S1

Relational Algebra Equivalence

Conjunctive selection operations can be deconstructed into a sequence of individual selections; cascade of σ

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

Only the final operation in a sequence of projection operations is needed, the rest can be omitted:

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) = \Pi_{L_1}(E)$$

Selections can be combined with Cartesian products and theta joins:

$$\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$$

$$\sigma_{\theta_1}(E_1 \bowtie_{\sigma_{\theta_2}} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$$

Selection & Theta-join & Natural-join operations are commutative:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

Theta joins are associative in the manner when θ_2 involves attributes from E_2 and E_3 only:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

The selection operation distributes over the theta join operation under the following two conditions:

1. It distributes when all the attributes in the selection condition σ_{θ_0} involves only the attributes of one of the expressions (E_1) being joined.
$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$
2. It distributes when the selection condition θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2
$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$

The projection operation distributes over the theta join.

a. Let L_1 and L_2 be attributes of E_1 and E_2 respectively. Suppose that the join condition θ involves only attributes in $L_1 \cup L_2$. Then
$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

b. Consider a join $E_1 \bowtie_\theta E_2$. Let L_1 and L_2 be sets of attributes from E_1 and E_2 respectively. Let L_3 be attributes of E_1 that are involved in the join condition θ, but are not in $L_1 \cup L_2$, and let L_4 be attributes of E_2 that are involved in the join condition θ, but are not in $L_1 \cup L_2$. Then
$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

The set operations union and intersection (*but not difference*) are commutative.

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

The selection operation distributes over the union, intersection, and set-difference operations.

$$\sigma_P(E_1 - E_2) = \sigma_P(E_1) - E_2 = \sigma_P(E_1) - \sigma_P(E_2)$$

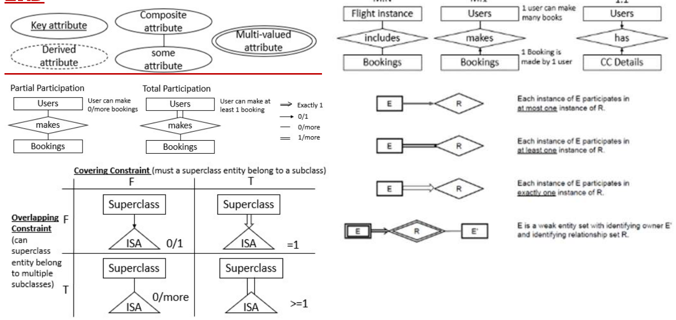
The projection operation distributes over the union operation.

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

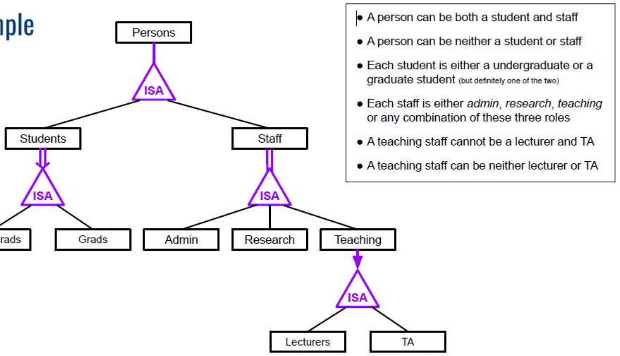
Relational Algebra Notes

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

ERD

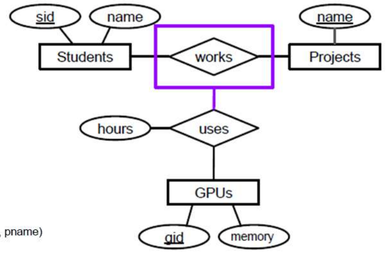


Example



Aggregates

```
CREATE TABLE Uses (
    gid      INTEGER,
    sid      CHAR(20),
    pname    VARCHAR(50),
    hours    NUMERIC,
    PRIMARY KEY (gid, sid, pname),
    FOREIGN KEY (gid) REFERENCES GPUs (gid),
    FOREIGN KEY (sid, pname) REFERENCES works (sid, pname)
);
```



SQL

Possible Actions for ON DELETE and ON UPDATE:

→ **NO ACTION:**	Reject delete/update if violates constraint (DEFAULT)
→**RESTRICT:**	Similar to NO ACTION except check cannot be deferred
→ **CASCADE:**	Propagates delete/update to referencing tuples
→ **SET DEFAULT:**	Updates FK of referencing tuples to some default value
→ **SET NULL:**	Updates FK of referencing tuples to NULL

Deferrable Constraints

Available for: UNIQUE, PRIMARY KEY, FOREIGN KEY

Example: *`CONSTRAINT manager_fk FOREIGN KEY (manager) REFERENCES Employees (id)`* *<u>`NOT DEFERRABLE`</u>*

→ **`NOT DEFERRABLE, DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INTIALLY IMMEDIATE`**

In TX: **`SET CONSTRAINTS … DEFERRED`**

Functions & Procedures

```
CREATE OR REPLACE FUNCTION <name>
(<param> <type>, <param> <type>, …)
RETURN <type> AS $$
    <CODE>
$$ LANGAUGE <language>
*where <language> includes sql or
plpgsql
```

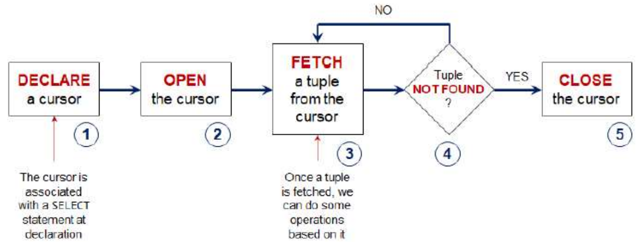
Return	Type
Single existing tuple	`<table_name>`
Set of existing tuple	`SET OF <table_name>`
Single new tuple	`RECORD`
Set of new tuple	`SET OF RECORD/TABLE(...)`
No return value?	*Use* `PROCEDURES` *or* `VOID`
Triggers	`TRIGGER` *(next topic)*

Control Structures

Variable	`DECLARE [<var> <type>]`$^+$ `<var> := <expr>`	
Selection	`IF … THEN … [ELSIF … THEN …]`* `[ELSE …] END IF`	
Repetition	`LOOP … END LOOP`	*EXIT … WHEN*
	`WHILE … LOOP … END LOOP`	*1…N*
	`FOR … IN … LOOP … END LOOP`	
Block	`BEGIN … END`	

Cursor

Declare → Open → Fetch → Check (repeat) → Close



Cursor Movement

```
FETCH curs INTO r
FETCH PRIOR FROM curs INTO r        //from previous row
FETCH FIRST FROM curs INTO r        //from top row
FETCH LAST FROM curs INTO r         //from last row
FETCH ABSOLUTE n FROM curs INTO R   //from nth row
```

Triggers

Trigger Functions

```
CREATE OR REPLACE FUNCTION <func>()
RETURNS TRIGGER AS $$
BEGIN
  <trigger_code>
END;
$$ LANGUAGE plpgsql;
```

Triggers

```
CREATE TRIGGER <name>
<trigger_timing>
  <trigger_event>
  ON <table>
FOR EACH <trigger_granularity>
  [ WHEN <condition> ]
  EXECUTE FUNCTION <func>();
```

Trigger Options

Events

- `INSERT ON <table>`
- `DELETE ON <table>`
- `UPDATE [ OF <column> ] ON <table>`

These set the `TG_OP` variable

Timings

- `AFTER/BEFORE` *(after or before the event)*
- `INSTEAD OF` *(replaces event, only for `VIEWS`)*

Granularities

- `FOR EACH ROW/FOR EACH STATEMENT`

BEFORE trigger is executed before the EVENT

- May modify the EVENT by using return value
- May cancel the EVENT by using return value

AFTER trigger is executed after the EVENT

- The EVENT already occurred, o cancellation cannot be done using return value
- Can still be done using **`RAISE EXCEPTION`**

`TG_OP = {'INSERT', "DELETE", 'UPDATE'}`

Triggers

Transition Variables

NEW: The modified row **after** the triggering event
OLD: The modified row **before** the triggering event

Events	NEW	OLD
INSERT	✓	✗
UPDATE	✓	✓
DELETE	✗	✓

Effect of Return Value

Events + Timings	NULL Tuple	NON-NULL Tuple t
BEFORE INSERT BEFORE UPDATE BEFORE DELETE	No insertion No update No deletion	Tuple t will be inserted/updated Deletion proceeds as normal *(t is ignored)*
AFTER INSERT AFTER UPDATE AFTE DELETE	NO EFFECT *(the operation is already done)*	

Trigger Granularities

Row-Level Trigger
FOR EACH ROW → Executes the trigger function for every tuple encountered
Statement-Level Trigger
FOR EACH STATEMENT → Executes the trigger for every statement regardless of the number of tuples

Timing	Row-Level	Statement-Level
AFTER	Tables	Tables and Views
BEFORE	Tables	Tables and Views
INSTEAD OF	Views	

Trigger Condition: WHEN

Example

```
CREATE TRIGGER bar
BEFORE INSERT ON Scores
FOR EACH ROW
  WHEN (NEW.StuName = 'Adi')
  EXECUTE FUNCTION foo();
```

Limitations

- *NO* SELECT in WHEN()
- *NO* OLD in WHEN() for INSERT
- *NO* NEW in WHEN() for DELETE
- *NO* WHEN() for INSTEAD OF

Deferred Trigger
Triggers that are checked only at the end of the TX instead of each statement

```
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>();
```

Activation Order
For the same event on the same table:

1. BEFORE statement-level triggers
2. BEFORE row-level triggers
3. EVENT for the given ROW
4. AFTER row-level triggers
5. AFTER statement-level triggers

If (2) returns NULL, then (3) and (4) are cancelled
Within category: alphabetical order

Functional Dependencies

If we keep every attribute in one table and do not enforce the FD, we can experience anomalies:

- Redundant storage — Update anomalies
- Deletion anomalies — Insertion anomalies

The purpose of normal forms is to recognize designs that enforce functional dependencies by means of main SQL constraints and thus protect data against anomalies.

- A functional dependency X → Y is **trivial** iff $Y \subset X$.
- A functional dependency X → Y is **completely non-trivial** iff $Y \neq \emptyset$ and $Y \cap X = \emptyset$.
- Let R be a relation. Let $S \subset R$ be a set of attributes of R. S is a **superkey** of R iff S → R.
- Let R be a relation. Let $S \subset R$ be a set of attributes of R. S is a **candidate** of R iff S → R and for all $T \subset S, T \neq S$, T is not a superkey of R.
- A **prime attribute** is an attribute that appears in some candidate key of R with Σ.
- Two sets of FD Σ and Σ′ are equivalent iff they have the same closure: $\Sigma \equiv \Sigma',\ \Sigma^+ = \Sigma'^+$

Let Σ be a set of FDs of a relational schema R. The closure of a set of attributes $S \subset R$, denoted S^+, is the set of all attributes that are functionally dependent on S. $\boldsymbol{S^+ = \{A \in R \mid \exists (S \to \{A\}) \in \Sigma^+\}}$

Armstrong Axioms

Reflexivity: $\forall X \subset R\ \forall Y \subset R\ \big((Y \subset X) \to (X \to Y)\big)$
Augmentation: $\forall X \subset R\ \forall Y \subset R\ \forall Z \subset R\big((X \to Y) \to (X \cup Z \to Y \cup Z)\big)$
Transitivity: $\forall X \subset R\ \forall Y \subset R\ \forall Z \subset R\ \big((X \to Y) \land (Y \to Z) \to (X \to Z)\big)$

Other Axioms:

Weak Reflexivity: $\forall X \subset R\ (X \to \emptyset)$
Weak Augmentation: $\forall X, Y, Z \subset R\ \big((X \to Y) \to (X \cup Z \to Y)$

Minimal Cover:

A set Σ of FD is **minimal** iff

1. The RHS of every FD in Σ is minimal. Every FD is of the form X → {A}.
2. The LHS of every FD is minimal. Every FD in Σ of the form X → {A} there is no FD Y → {A} in Σ^+ s.t. Y is a proper subset of X.
3. The set itself it minimal. None of the FD in Σ can be derived from other FD in Σ.

A **minimal cover** of a set of FD Σ is set of FD Σ′ that is both minimal and equivalent to Σ.

A set of FD is **compact** iff there is no different dependencies with same LHS.
{{A} → {B}, {A} → {C}} is not compact. {{A} → {B,C}} is compact.

A **compact cover** of a set of FDs Σ is set of FDs Σ′ that is both compact and equivalent to Σ. A compact minimal cover of a set of FDs Σ is set of FDs Σ′ that is both compact, minimal and equivalent to Σ.

BCNF

A relation R with a set of FD Σ is in BCNF ↔ ∀ FD $X \to \{A\} \in \Sigma^+$:

- $X \to \{A\}$ is trivial or
- X is a superkey

It is sufficient to look at Σ.

Decomposition

A **decomposition** of a table R is a set of tables $\{R_1, \dots, R_n\}$ s,t $R = R_1 \cup \dots \cup R_n$.
A **binary decomposition** of a table R is a pair of tables $\{R_1, R_2\}$ s.t. $R = R_1 \cup R_2$. A binary decomposition is loseless-join iff the full outer natural join of its two fragments equals the initial table.

Loseless-Join

A binary decomposition of R into R_1 and R_2 is **loseless-join** if $R = R_1 \cup R_2$ and $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$.
If $R_1 \cap R_2$ is the PK of one of the two tables, then it can be a FK in the other referencing the PK.

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

Dependency Preserving

A decomposition of R with Σ into $R_1 \dots R_n$ with the respective sets of projected FD $\Sigma_1 \dots \Sigma_n$ is dependency preserving iff $\Sigma^+ = (\Sigma_1 \cup \dots \cup \Sigma_n)^+$.

Decomposition Algorithm

Let X → Y be a FD in Σ that violates the BCNF definition (it is not trivial and X is not a superkey). We use it to decompose R into the following two relations R_1 and R_2.

- $R_1 = X^+$
- $R_2 = (R - X^+) \cup X$

We must now check whether R_1 and R_2 with the respective sets of projected FD Σ_1 and Σ_2 are in BCNF and <u>continue</u> the decomposition if they are not. The decomposition algorithm is guaranteed to find a **loseless decomposition in BCNF**, but may <u>not</u> be dependency preserving.

3NF

A relation R with a set of FDs Σ is in 3NF ↔ every FD $X \to \{A\} \in \Sigma^+$

- $X \to \{A\}$ is trivial or
- X is a superkey or
- A is a prime attribute

It is sufficient to look at Σ.

Bernstein Algorithm

When a relation is not in 3NF, we can synthesis a schema in 3NF from a **minimal cover** of the set of FDs.

- For each FD X → Y in the minimal cover create relation $R_i = X \cup Y$ unless it already exists or is subsumed by another relation.
- If none of the created relations contains 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 a compact minimal cover. The algorithm guarantees a **loseless, dependency preserving decomposition in 3NF**.