CS2102 Database Systems
by Lee Zong Xun

Relational Model

Key Constraints

- **Superkey** $\to$ subset of attributes that *uniquely* identifies a tuple in a relation. Attributes A is not a superkey if there are two tuples t_1 and t_2 such that
  - $\forall a \in A : t_1.a = t_2.a \wedge \exists b \in R - A : t_1.b \neq t_2.b$
- **Key** $\to$ superkey that is **minimal**
- **Primary key** $\to$ selected candidate key for a relation
  - *must* be non-`NULL` and `UNIQUE` **(entity integrity constraint)**
  - Candidate keys may be `NULL` or contain `NULL` columns.

Foreign Key Constraints

- A **foreign key** is a subset of attributes of relation $R1$ that
  1. appear as a **primary key** in the referenced relation $R2$
  2. be a `NULL` value (OR contain at least one `NULL` value)

Relational Algebra

Relations are a **set**. Relations are closed under Relational Algebra.

Conjunction

$c_1 \wedge c_2$		c_1: False	c_1: NULL	c_1: True
c_2	False	False	False	False
	NULL	False	NULL	NULL
	True	False	NULL	True

Disjunction

$c_1 \vee c_2$		c_1: False	c_1: NULL	c_1: True
c_2	False	False	NULL	True
	NULL	NULL	NULL	True
	True	True	True	True

Binary Operators

Set Operators

Two relations are **union-compatible** if
- R and S have the same number of attributes;
- corresponding attributes have the *same or compatible domains*; and no requirement for same attribute names

Cross product

$R \times S$ returns a relation with schema (A, B, X, Y) where $R \times S = \{(a, b, x, y) \mid (a, b) \in R, (x, y) \in S\}$
- $|R \times S| = |R| * |S|$ (note **empty tables**)
- Attributes A and B **must be disjoint**

Division

Consider two relations $R(A_1, \ldots, A_m)$, $S(B_1, \ldots, B_n)$, where $attr(S) \subset attr(R)$. The division of R by S computes the largest set of tuples $Q \subset \pi_{[A_1,\ldots,A_m]}(R)$ such that for every tuple $(a_1, \ldots, a_m) \in Q$,

$$\pi_{[L]}((a_1, \ldots, a_m) \times S) \in R$$

R/S is equivalent to $\pi_{[A]}(R) - \pi_{[A]}((\pi_A(R) \times S) - R)$

Join operators

Inner Joins

- **Natural Join** $\bowtie$ Let A be the set of attributes that R and S have in common, $c = \forall a_i \in A : R_{a_i} = S_{a_i}$ and $\ell = Attr(R) \cup (Attr(S) - A)$, then

$$R \bowtie S = \pi_\ell(R \bowtie_c S)$$

Outer Joins

- ***dangle***$(R \bowtie_\theta S)$ $\to$ set of dangling tuples in R wrt to $R \bowtie_\theta S$ — *dangle*$(R \bowtie_\theta S) \subseteq R$
- **null**(R) $\to$ n-component **tuple** of `null` values where n is the number of attributes of R

Types

- **Left outer join**

$$R_\theta S = R \bowtie_\theta S \cup \textit{dangle}(R \bowtie_\theta S) \times \{\textit{null}(S)\}$$

- **Right outer join**

$$R_\theta S = R \bowtie_\theta S \cup (\{\textit{null}(R)\} \times \textit{dangle}(S \bowtie_\theta R))$$

- **Full outer join**

$$R_\theta S = R \bowtie_\theta S \cup (\textit{dangle}(R \bowtie_\theta S) \times \{\textit{null}(S)\})$$
$$\cup(\{\textit{null}(R)\} \times \textit{dangle}(S \bowtie_\theta R))$$

- **Natural Outer join** is analogous to natural inner join where equality operation is performed over all attributes that R and S have in common.

Equivalence

- **Strongly** equivalent if for any input **both produces error** or **both produces the same result**.
- **Weakly** equivalent if for any input there is no error then both produces the same result.
- $\sigma_{[c]}(A \cup B) \not\equiv \sigma_{[c]}(A) \cup \sigma_{[c]}(B)$ (invalid columns).
- $\pi_{[A]}(R - S) \not\equiv \pi_A(R) - \pi_A(S)$ (e.g. 12, 13).
- $\pi_{[D,Y]}(R \times S) \equiv \pi_{[D,Y]}(S \times R) \equiv \pi_{[D]}(S) \times \pi_{[Y]}(R)$
- $R \bowtie_{[\theta_1]} (S \bowtie_{[\theta_2]} T) \not\equiv (R \bowtie_{[\theta_1]} S) \bowtie_{[\theta_2]} T$ unless θ_1 uses T and θ_2 uses R.
- $(E_1 \bowtie_{c1} E_2) \bowtie_{c2 \wedge c3} E3 \not\equiv E_1 \bowtie_{c1 \wedge c2} (E_2 \bowtie_{c3} E3)$
- $\sigma_{[\theta]}(R \times S) \not\equiv \sigma_{[\theta]}(R) \times S$

SQL Overview

Integrity Constraints

Unique constraint

The following expression should hold at all times:

$$\forall r_1, r_2 : (r_1 \equiv r_2) \vee (\exists a : r_1.a <> r_2.a)$$

Since the *unique* check is done via <>, as long as the tuple in question contains a `NULL` value, then the condition always evaluates to `NOT false`.

Deferrable Constraints

Constraints are checked immediately at the end of a SQL statement. This is true even for **transactions**.
- Available for `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`.

```
ALTER TABLE <TBL>
ADD CONSTRAINT <NAME>
FOREIGN KEY(Attr) REFERENCES TBL(Attr)
DEFERRABLE INITIALLY DEFERRED | IMMEDIATE;
```

Considerations

- No need to care about the order of statements inside a transaction
- Allows for cyclic foreign key constraints
- Data definition no longer unambiguous
- Certain checks need to be done at run-time (performance penalty).

Querying the database

By default, PostgreSQL allows duplicate tuples to exist, and row-ordering is not guaranteed (order independent).

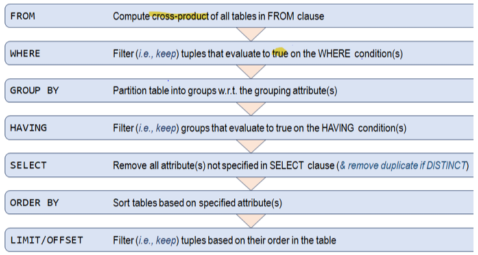
Common SQL Constructs

Grouping

Given `GROUP BY a1, ...` two tuple t and t' belong to the same group if $r1.a1 \equiv r2.a1, \ldots$ If column A_i of **table R** appears in the SELECT/HAVING clause, either
1. A_i appears in the GROUP BY clause
2. A_i appears as input of an aggregation function in the SELECT/HAVING clause
3. The primary key of R appears in the GROUP BY clause

Remark:
1. There should be unique values for each group. If not, then the result is undefined.
2. Group based on **equivalence** (incl. NULL)
3. If the `SELECT` clause has multiple attributes from different tables, the same condition applies to each individual attribute.

Conceptual Evaluation of Queries



Conditional expressions

case statements: Very similar to how a normal switch-case statement is written.

```
-- preferred
CASE [expression]
  WHEN <condition | value> THEN <result>
  WHEN <condition | value> THEN <result>
  ELSE <default>
END
```

Structuring SQL Queries

Views

Only parts of the table is of interest. Equivalent to an external schema that is presented to the user.

Note:
- A `VIEW` is a permanently named query. The computation may be done each time the virtual table is accessed (i.e. not permanently stored).
- Can be used as a normal table.
- For `INSERT`, `UPDATE`, and/or `DELETE` statements.
  1. Only one entry in the `FROM` clause
  2. No `WITH DISTINCT, GROUP BY, LIMIT or OFFSET`
  3. No `UNION, INTERSECT, EXCEPT or ALL`
  4. No aggregates

Extended Concepts

Recursive Queries

```
WITH RECURSIVE Linker(to_stn, stops) AS (
  SELECT to_stn, 0
  FROM MRT WHERE fr_stn = "NS1"
  UNION [ ALL ]
  SELECT M.to_stn, L.stops + 1
  FROM Linker L, MRT M
  WHERE M.fr_stn = L.to_stn
  AND L.stops < 3 -- termination
)
SELECT to_stn, stops FROM Linker;
```

General Guidelines

Relation cardinality

Suppose entities A and B has a relationship R. The following table shows the cardinality of the relationship R.

Relationship	Min	Max
(total)-to-(many)	A	AB
(total)-to-(key) $\to$ error if $A > B$	A	B
(key)-to-(many)	0	A
(total+key)-to-(many)	A	A
(total)-to-(total)	max(A, B)	AB
(key)-to-(key)	0	min(A, B)
(many)-to-(many)	0	AB
(total+key)-to-(total) $\to$ error if $A < B$	A	A
(total+key)-to-(key) $\to$ error if $A > B$	A	A
(total+key)-to-(total+key) $\to$ error if $A <> B$	A=B	A=B

Procedures and Functions

Statment level interfaces allows us to execute SQL in a host language. **Call level interfaces** allows us to code in the host language only.

Functions

Functions are reusable snippets that **return some value**. General syntax:

```
CREATE OR REPLACE FUNCTION
  <NAME>(<IN|OUT|INOUT> ARGS <TYPE>)
RETURNS <SQL_RETURN_TYPE | [SETOF] TBL_NAME |
    RECORD | SETOF RECORD> AS $$
:
$$ LANGUAGE sql
```

Procedures

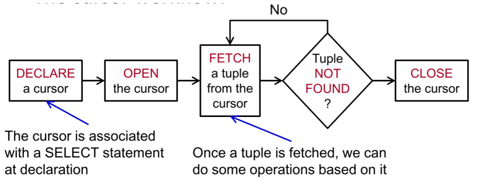
Procedures are reusable snippets that **do not return any value** i.e. update. They are transactional.
General syntax:

```
CREATE OR REPLACE PROCEDURE <NAME>(ARGS <TYPE>)
AS $$
:
$$ LANGUAGE sql;

CALL PROC()
```

Cursors

Cursor workflow:



For example:

```
CREATE OR REPLACE FUNCTION score_gap()
RETURNS TABLE ( name TEXT, mark INT, gap INT )
AS $$ DECLARE
  -- declares a cursor variable
  curs CURSOR FOR (SELECT * FROM Scores ORDER
      BY Mark DESC);
  r RECORD;
  prv_mark INT;
BEGIN
  prv_mark := -1;
  OPEN curs;
  LOOP
    FETCH curs INTO r; -- read the input into r
    EXIT WHEN NOT FOUND; -- terminate
    name := r.Name;
    mark := r.Mark;
    IF prv_mark >= 0 THEN
      gap := prv_mark - mark;
    ELSE gap := NULL;
    END IF;
    RETURN NEXT; -- inserts tuple to output
    prv_mark := r.Mark;
  END LOOP;
  CLOSE curs; -- releases resources
END;
$$ LANGUAGE plpgsql;

FETCH -- some variants
[PRIOR | FIRST | LAST | ABSOLUTE NUM]
FROM cur INTO r
```

SQL Injection Attack

Use prepared statements instead of directly executing SQL.

```
-- instead of using this
EXEC SQL EXECUTE IMMEDIATE :stmt;

-- use this
EXEC SQL PREPARE mystmt FROM :stmt;
EXEC SQL EXECUTE mystmt USING :in_name;
```

Triggers

```
-- Trigger
CREATE TRIGGER <NAME>
<AFTER | BEFORE | INSTEAD OF> <INSERT | DELETE
    | UPDATE> ON <TBL>
[ WHEN <condition>]
FOR EACH <ROW | STATEMENT> EXECUTE FUNCTION <
    trigger_func>();

-- Trigger function
CREATE OR REPLACE FUNCTION <NAME>()
RETURNS TRIGGER AS $$
BEGIN
  :
END;
$$ LANGUAGE plpgsql;
```

A trigger function has access to a number of variables.

- `NEW` — the new tuple
- `OLD` — the old tuple (update/delete)
- `TG_OP` — the operation that activates the trigger
- `TG_TABLE_NAME` — name of table that trigger the invocation
- `CURRENT_DATE` — the current date

Notes

- Returning `NULL` in a `BEFORE` trigger tells the database to ignore the rest of the operation.
- `OLD` is set to NULL in a `BEFORE INSERT` trigger.
- Whenever the function returns a non-null tuple, the function will use it as the tuple to be inserted (incl. OLD).
- For `BEFORE` trigger, returning a non-null value will cause the operation to proceed as normal. Otherwise, no tuples will be inserted.
- The return value for an `AFTER` trigger is ignored.
- For `INSTEAD OF` trigger, returning non-null signals to proceed as normal, whereas null indicates to ignore the rest of the operations on the **current** row.
- `Instead Of` triggers can only be used on views.

```
CREATE TRIGGER update_max_trigger
INSTEAD OF UPDATE ON Max_Score -- view
FOR EACH ROW EXECUTE FUNCTION update_max_func
    ();

CREATE OR REPLACE FUNCTION update_max_func()
RETURNS TRIGGER AS $$ BEGIN
  -- actual table
  UPDATE Scores SET Mark = NEW.Mark WHERE
      Name = OLD.Name;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Trigger levels

Statement-level triggers ignore the values returned by the trigger function. To omit subsequent operations, raise an **exception**.

Trigger timing

`INSTEAD OF` is only allowed on row-level.

Trigger condition

Condition in WHEN() can be more complicated subject to the following restrictions:

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

Deferred Trigger

Defer the checking of triggers until the end of the transaction.

```
CREATE CONSTRAINT TRIGGER bal_check_trigger
AFTER INSERT OR UPDATE OR DELETE ON Account
DEFERRABLE INITIALLY DEFERRED
-- must be for each row
FOR EACH ROW EXECUTE FUNCTION bal_check_func();

-- if set INITIALLY IMMEDIATE
BEGIN TRANSACTION;
SET CONSTRAINTS bal_check_trigger DEFERRED;
:
COMMIT;
```

Order of triggers

Within each category, triggers are activated in alphabetic order. If a BEFORE row-level trigger returns NULL, then subsequent triggers on the same row are omitted.

1. BEFORE statement-level
2. BEFORE row-level
3. AFTER row-level
4. AFTER statement-level

Functional Dependencies

Two attributes are functionally equivalent if their closures are equal. All candidate keys are functionally equivalent.

- $\alpha \rightarrow \beta$ is a **trivial** FD if $\beta \subset \alpha$.
- $\alpha \rightarrow \beta$ is a **non-trivial** FD if $\beta \not\subset \alpha$.
- $\alpha \rightarrow \beta$ is a **completely non-trivial** FD if $\beta \cap \alpha = \emptyset$. Both A and B cannot be empty sets.

Computing attribute closures

A **closure** is a set of all attributes that are functionally dependent on a given set of attributes.

- Check if any attribute doesn't appear in the RHS of any FD. These attributes must appear in the key
- Compute attribute closure starting with singular attributes.
- Note all candidate keys in the process
- If current set of attributes is a superset of some previously seen, candidate key, can skip

Anomalies

- **Insertion Anomaly** — Unable to insert a record into a table because the table's structure does not allow us to enter certain information.
- **Update Anomaly** — Update one instance of the data without updating all other instances.
- **Deletion Anomaly** — Loss of some, or all, of the information related to the deleted record.

Armstrong Axioms

Sound: The rule only generates elements of Σ^+ when applied to Σ **Complete**: The rule(s) generate(s) all elements of Σ^+ when applied to Σ

- $XY \rightarrow Y$ (Reflexivity)
- $X \rightarrow Y \Rightarrow XZ \rightarrow YZ$ (Augmentation)
- if $X \rightarrow Y$ and $Y \rightarrow Z \Rightarrow X \rightarrow Z$ (Transitivity)
- if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$ (Union)
- if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$ (Decomposition)
- if $A \rightarrow B$ and $BC \rightarrow D$, then $AC \rightarrow D$ (Pseudo-transitivity)
- if $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$ (Composition)
- if $AC \rightarrow BC$, then $A \not\rightarrow B$ (Reverse augmentation)
- if $X \rightarrow Y$, then $XZ \rightarrow Y$ (Weak augmentation)

Normalization

BCNF

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

In general, a violation occurs, iff we have a non-trivial closure that contains **more** attributes than the left hand side of the FD, but does not contain all the attributes in the table.

BCNF Algorithm

1. Find a subset X of the attributes in R s.t. its closure is a violation of BCNF.
2. Decompose R into two tables R_1 and R_2 s.t. R_1 contains $\{X\}^+$ and R_2 contains $(R - \{X\}^+) \cup X$.
3. If R_1 or R_2 is not in BCNF, decompose them further.

Lossless join

The original table can always be reconstructed from the decomposed tables, whenever the common attributes in R_1 and R_2 constitutes a superkey in R_1 **or** R_2.

- A binary decomp is lossless-join $\iff$ full outer natural join of its two fragments equal the initial table. Otherwise it is lossy
- A binary decomp of R into R_1 and R_2 is lossless-join if $R = R_1 \cup R_2$ and either $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$
- A decomp is lossless-join if there exists a sequence of binary lossless-join decomp that generates that decomp

Dependency preserving

A decomp of R with Σ into $R_1, R_2, \cdots, R_n$ with respective projected FDs $\Sigma_1, \Sigma_2, \cdots, \Sigma_n$ is dependency preserving $\iff \Sigma^+ = (\Sigma_1 \cup \cdots \cup \Sigma_n)^+$

- Check that the minimal cover are equivalent

Notes

- The BCNF decomposition of a table may not be unique.
- If a table has only 2 attributes, then it must be in BCNF.
- Each decomposition step gets rid of at least one BCNF violation $\implies$ termination is guaranteed.
- If there are no FDs in a table, the table is already in BCNF $\implies$ key of the table consists of all the attributes in the table.
- No update, deletion and insertion anomalies.
- Cannot derive original FDs from R_1 and R_2 $\implies$ dependencies may not be preserved (not equivalent).

3NF

A table is in 3NF iff for every non-trivial and decomposed FD either (1) the left hand side is a superkey, or (2) right hand side is a proper subset of a candidate key. 3NF is not as strict as BCNF $\implies$ higher levels of redundancy.

3NF Decomposition Algorithm

Given: A table R, and a set S of FDs

1. Step 1: Derive a minimal basis of S
2. Step 2: In the minimal basis, combine the FDs whose left hand sides are the same
3. Step 3: Create a table for each FD remained
4. Step 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. Step 5: Remove redundant tables

The **minimal basis** of S is a simplified version of S, such that (1) every FD in S can be derived from M, (2) every FDs in M is non-trivial and decomposed FD, (3) no FD in M is redundant and (4) none of the attributes on the LHS are redundant.

Algorithm for Minimal Basis

- Transform the FDs, so that each right hand side contains only one attribute.
- Remove redundant attributes on the left hand side of each FD.
- Remove redundant FDs.

May have update and delete anomalies in rare cases.