CS2102 Database Systems

by Lee Zong Xun

Database Management System

Transactions

A finite sequence of *DB* operations, constitutes the smallest logical unit of work from an application perspective and adheres to **ACID** properties. → Performing transactions yields *a sequence of consistent states* in the process.

ACID properties

1. **Atomicity**: All or nothing
2. **Consistency**: Execution of T guarantees to yield a *correct state* of the DB
3. **Isolation**: Execution of T is *isolated* from the effects of concurrent transactions
4. **Durability**: Committing T ensures that its effects are *permanent* even in case of failures

Serialization

Serial

less performant without optimizations but yields the correct result.

Concurrent

associated problems like lost update / dirty read / unrepeatable read.

Serializability

A concurrent execution is **serializable** if this execution is **equivalent** to some serial execution of the same set of transactions.

- **equivalent** → they have the same *effect* on the data

Core tasks of DBMS

- Support *concurrent executions* of transactions to optimise performance
- enforce *serializability* of concurrent executions to ensure integrity of data

Architecture

A 3-layer abstraction consisting of:

- **External (User view on the data)**
- **Logical (Organization of data)**
  - Logical independence — change without affecting external (e.g. change data types, model or adding new attributes)
  - Physical independence — representation of data is independent of the physical schema
- **Physical (Organization of data on disk and in memory)**

Relational Model

Unified representation of all data as relations

Definitions

- **Domain** → a set of *atomic* values
  - Each value v of attribute A_i is either: $v \in dom(A_i)$ or $v = \texttt{NULL}$
- **Relation** → a set of *tuples*
  - Each instance of schema R is a subset of $\{(a_1, a_2, \ldots, a_n) \mid a_i \in dom(A_i) \cup \{null\}\}$
- **Relation schema** → specifies the **attributes** and data constraints
- **Relational database schema** → set of relation schemas and data constraints
- **Relational database** → collection of tables

Data Integrity

- **Integrity constraint** → condition that restricts what constitutes valid data
- **Structural constraint** → inherent to the data model and independent of the application.
  - Domain, Key, and Referential constraints

Key Constraints

- **Superkey** → 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** → superkey that is **minimal**
- **Primary key** → 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)

Considerations

- Integrity constraints are optional. Allows check to be done by the DBMS, instead of the application. Ensures that no matter who uses the database, entries will be consistent with respect to the constraints.
- Integrity constraints may affect performance.

Relational Algebra

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

Conjunction

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

Disjunction

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

Unary Operators

Selection, σ_c

$\sigma_c(R)$ selects tuples from a relation satisfying condition c.

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

- No change to schema

Projection, π_ℓ

- $\pi_\ell(R)$ projects attributes of a given relation specified in the **ordered** list ℓ
  - Duplicate tuples are removed from the output relation
  - Resulting schema is modified

Renaming, ρ_ℓ

$\rho_\ell(R)$ renames attributes of a relation R.

- Resulting schema is modified

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***$(R \bowtie_\theta S)$ → set of dangling tuples in R wrt to $R \bowtie_\theta S$ — *dangle*$(R \bowtie_\theta S) \subseteq R$
- **null**(R) → 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**

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

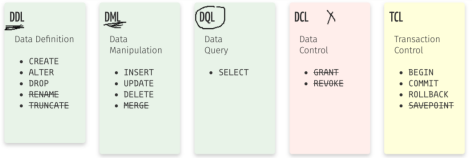
- **Natural Outer join** is analogous to natural inner join where equality operation is performed over all attributes that R and S have in common.

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).
- Selection is commutative and distributive over set operations.
- Selection distributes over $\cup, \cap, -$.
- $\pi_{a_1}(\pi_{a_2}(A)) \not\equiv \pi_{a_1}(A)$.
- $\pi_{[A]}(R - S) \not\equiv \pi_A(R) - \pi_A(S)$ (e.g. 12, 13).
- $\pi_{[D,Y]}(R \times S) \equiv \pi_{[D,Y]}(S \times R) \equiv \pi_{[D]}(S) \times \pi_{[Y]}(R)$
- Projection distributes over union
- Cross and Joins are not commutative, but are associative.
- $R \bowtie_{[\theta_1]} (S \bowtie_{[\theta_2]} T) \not\equiv (R \bowtie_{[\theta_1]} S) \bowtie_{[\theta_2]} T$ unless θ_1 uses T and θ_2 uses R.
- $(E1 \bowtie_{c1} E2) \bowtie_{c2 \wedge c3} E3 \not\equiv E1 \bowtie_{c1 \wedge c2} (E2 \bowtie_{c3} E3)$
- **Projection after selection** is not equivalent to **selection after projection** unless θ uses only attributes in the projection.
- $\sigma_{[\theta]}(R \times S) \not\equiv \sigma_{[\theta]}(R) \times S$

SQL Overview

Types of Commands



Data types

Basic Data Types

Type	Description
BOOLEAN	Logical Boolean (*true/false*)
INTEGER (INT)	Signed 4-bytes integer
FLOAT8	Double precision 8-bytes floating-point
NUMERIC[(p,s)]	Exact numeric of selectable precision
CHAR(n)	Fixed-length character string
VARCHAR(n)	Variable-length character string
TEXT	Variable-length character string
DATE	Calendar date (*year, month, day*)
TIMESTAMP	Date and time

Extended Data Types

Kinds	Types
Document	XML JSON
Spatial	Point Line Polygon Circle Box Path
Special	Money/Currency MAC/IP Address

User-Defined Types

```
CREATE TYPE
```

Main principles

- **Principle of acceptance** — perform the **operation** if the condition *evaluates to true* (used in `WHERE` clause).
- **Principle of rejection** — reject the **insertion** if the condition *evaluates to false* (used in integrity constraints).

Syntax

Table syntax

```
CREATE TABLE <tbl_name> (
  <attr1> <type1> [<col_constraints>]
  :
  [<tbl_constraints>] -- no comma
);

ALTER TABLE <tbl_name>
  ALTER COLUMN <col_name>
  [TYPE <type> | SET DEFAULT <value> |
  DROP DEFAULT];

ALTER TABLE <tbl_name>
  [DROP | ADD] CONSTRAINT [name | eid_fk
  FOREIGN KEY (eid) REFERENCES employee(eid)
      ];

ALTER TABLE <tbl_name>
  [ADD | DROP] COLUMN <col_name> <type>?;
```

Note: Name of constraint need to be known (can be retrieved from metadata).

```
DROP TABLE
  [IF EXISTS] -- no error if table not exist
  <table_name>[, <table_name> [...]] --
       multiple table
  [CASCADE]; -- also delete referencing table
```

Row syntax

```
INSERT into <tbl_name> [(attr1, attr_2, ...)]
  VALUES
    (<val_1>, <val_2>, ...),
    :
    (<val_1>, <val_2>, ...);
```

Some notes:

1. Either all or none are inserted.
2. Attributes can be specified out of order.
3. Missing values are replaced with NULL values.

```
UPDATE <tbl> SET <attr> = <value>
  [WHERE <condition>];
DELETE FROM <tbl_name> [WHERE <condition>];
```

If the condition is not specified, then defaults to true. Otherwise, can be arbitrarily complex.

Integrity Constraints

NOT NULL constraint

Violated if `<attr> IS NOT NULL` evaluates to `false`.

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`.

Primary key constraint

Primary key must be UNIQUE and NOT NULL.

Foreign key constraint

```
pname TEXT DEFAULT "FastCash",
:
PRIMARY KEY(eid, pname)
FOREIGN KEY(pname) REFERENCES Projects(name
    )
```

In most DBMS, a foreign key can point to any attribute with a UNIQUE constraint, not just a *primary key*. How can we deal with NULL values? Just add a NOT NULL constraint!

```
ON UPDATE <action> | ON DELETE <action>
```

To specify the behaviour when data in referenced table is deleted or updated using optional specification.

Keyword	Action
NO ACTION	*Reject* delete/update if it violates constraint *(default value)*
RESTRICT	Similar to "NO ACTION" except that check of constraint *cannot be deferred (deferrable constraints are discussed in a bit)*
CASCADE	*Propagates* delete/update to the referencing tuples
SET DEFAULT	*Updates* the foreign key of the referencing tuples to some default value *(Important: default value must be a primary key in the referencing table)*
SET NULL	*Updates* the foreign key of the referencing tuples to NULL value *(Important: corresponding column must be allowed to contain NULL values)*

Key considerations:

1. SET NULL issue with prime attributes.
2. SET DEFAULT issue with default value not in referenced relation.
3. CASCADE may create a chain of propagation, affecting **performance**.

General constraint

CHECK constraint

- Not a **structural integrity constraint**
- Allows us to specify that column values must satisfy an arbitrary boolean expression.
- **Scope**: one table, one row.

```
CREATE TABLE Projects (
  eid INT,
  pname TEXT,
  hours INT CHECK (hours > 0),
  PRIMARY KEY (eid, pname),
  CHECK (
      (pname = 'CoreOS' AND hours >= 30) OR
      (pname <> 'CoreOS' AND hours > 0)
    )
);
```

ASSERT constraint

- Used for arbitrary constraints (multiple table and rows)
- Have various potential side-effects/or limitations:
  - Cannot modify date, No proper error handling, Not linked to a specific table. Alternative: **triggers**

Deferrable Constraints

Constraints are checked immediately at the end of a SQL statement. This is true even for **transactions**.

Want: relax checks to be done at the **end** of a transaction instead → the constraints may be *temporarily* violated.

- Available for UNIQUE, PRIMARY KEY, FOREIGN KEY.

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

```
ALTER TABLE Employee
  ADD CONSTRAINT did_fk
  FOREIGN KEY(did) REFERENCES Department(did)
  DEFERRABLE INITIALLY IMMEDIATE;

ALTER TABLE Department
  ADD CONSTRAINT eid_fk
  FOREIGN KEY (eid) REFERENCES Employee (eid)
  DEFERRABLE INITIALLY IMMEDIATE;

BEGIN; -- start of transaction
SET CONSTRAINTS did_fk DEFERRED;
INSERT INTO  -- did_fk needs to be deferred
  Employee VALUES (101, 'Sarah', 1001);
INSERT INTO -- but not eid_fk
  Department VALUES (1001, 'dev', 101);
COMMIT; -- successful end of transaction
```

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).

Entity Relationship model

All data is described in terms of **entities** and their **relationships**.

Definitions

- **Entity set** → collection of entities of the same type
- **Attribute** → specific information describing an entity
  - **Key attribute** → uniquely identifies each entity
  - **Composite attribute** → composed of multiple other attributes (oval of ovals)
  - **Multivalued attribute** → may comprise more than one value for a given entity (double-lined oval)
  - **Derived attribute** → derived from other attributes (dashed oval)
- **Relationship set** → collection of relationships of the same type
  - In a self-referential relationship, an entity type is related to instances of itself, either directly or indirectly.

Cardinality Constraints

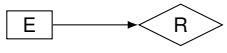
- Describes an **upper bound** to the number of times an entity can participate in a relationship (either 1 or ∞)
- Upper limits of 1 are called **key constraints**

Participation Constraints

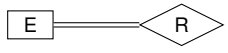
- Describes a **lower bound** to the number of times an entity can participate in a relationship (either 0 or 1)
- **Partial participation constraint**: optional participation
- **Total participation constraint**: mandatory participation

Dependency Constraints

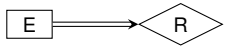
- **Weak entity sets**
  - Its own key is called a **partial key**.
    - Cannot uniquely identity entity
    - Unique only with the help of PK from **owner entity**
  - Existence depends on the existence of its owner entity
  - Connected via an **identifying relationship** (not represented on SQL schema)
- Requirements
  1. many-to-one relationship (identifying relationship) from weak entity set to owner entity set
  2. weak entity set must have **total participation** in identifying relationship



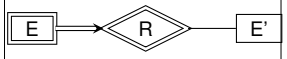
each instance of E participates in **at most one** instance of R



each instance of E participates in **at least one** instance of R



each instance of E participates in **exactly one** instance of R



E is a **weak entity set** with identifying owner E' & identifying relationship set R.

Relational Mapping

Assume data type is as logical as possible.

To handle composite/multivalued attributes,

1. convert to single-valued columns (attributes)
2. additional table with FK constraint (not recommended)
3. convert to VARCHAR (concat the values)

Guidelines

- "$p \implies q$" is equivalent to "$(\tilde{p}) \vee q$".
- Make use of FK and NOT NULL constraint to enforce $\implies$ constraint. This can also be enforced by using a UNIQUE constraint.
- Derived attributes should not appear in the logical schema.
- ISA contraints cannot be represented in SQL
- Attributes should have the same name if and only if they are semantically equivalent (universal schema assumption)

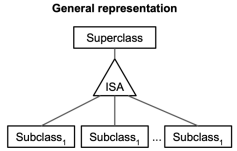
Extended Notations

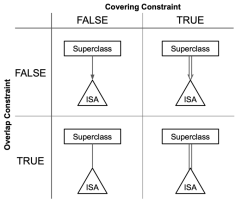
ISA Hierarchy

- Every entity in a subclass is an entity in its superclass
  - Superclass and subclass must be uniquely identified by the same key
  - Subclass key should not be shown in the ER diagram

Constraints

- **Overlap contraint** → a superclass entity can belong to **multiple** subclasses
- **Covering constraint** → a superclass entity **has to** belong to a subclass





Aggregation

Relationships only between entity sets, not between relationship sets.

- Abstraction that treats relationships as higher-level entities
  - Aggregate is rectangle (i.e. entity sets) around the diamond (i.e. relationship sets)
  - The rectangle **should not** touch the diamond

Querying the database

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

Syntax

Use `IS NULL` to check for NULL values.

SELECT

```
SELECT [ DISTINCT ] <target-list>
FROM <relation-list>
[ WHERE <condition> ];
```

The above query takes the cross product of the lists specified, filters the tuples based on some condition and return only the specified columns.

- Tuples are distinct if $\exists a \in R : a1$ IS DISTINCT FROM $a2$.
- IS DISTINCT FROM is equivalent to "$\neq$"
- Table can be used multiple times only if an alias is used.

Operations

- Mathematical: +, *, −, ⁄, /, ^, |/, *etc*
- String: || *(concatenate)*, LOWER(s), UPPER(s), *etc*
- Date Time: +, NOW(), *etc*

Renaming

- column AS alias

WHERE

Use LIKE for basic pattern matching.

```
WHERE pizza [ NOT ] LIKE 'Ma%a';
```

- '%' matches 0 or more characters
- '_' matches any single characters

ORDERING

```
ORDER BY <attributes> ASC/DESC
OFFSET 3
LIMIT 3;
```

Operations

Set operations

```
(Query 1)
UNION|EXCEPT|INTERSECT
(Query 2)
```

`UNION, INTERSECT, EXCEPT` will eliminate duplicate tuples from the result. `UNION ALL, INTERSECT ALL, EXCEPT ALL` treats each element in a relation as **distinct**, but not across relations.

JOIN operations

The **universal relation assumption** states that one can place all data attributes into a table, which may then be decomposed into smaller tables as needed.

But be wary of tables that do not satisfy this assumption. You cannot join without checking for relevancy.

Note: Only natural joins remove common attributes

Subqueries

Can appear in `SELECT, FROM, WHERE` clause. Common subqueries expressions include: `IN/NOT IN, EXISTS/NOT EXISTS, ANY and ALL`.

- Either returns **at most** a single value or **exactly one** column.
- If result returns 0 rows, then it is treated as `NULL`.

Semantics

- `IN` returns TRUE if $\exists row \in$ subquery $: row = expr$
- `NOT IN` returns TRUE if $row <> expr$ $\forall row \in$ subquery
- `ANY` returns TRUE if comparison is TRUE for at least one row in the subquery.
- `ALL` returns TRUE if comparison is TRUE for ALL rows in the subquery.
- `EXISTS` returns TRUE if there is at least one row
- `NOT EXISTS` returns TRUE if subquery returns empty

Scoping

- A table alias declared in a (sub-)query Q can only be used in Q or subqueries nested within Q
- If the same table alias is declared in both subquery Q1 and in outer query Q0 (or not at all), the declaration in Q1 is applied

Remarks

Not all constructs are absolutely required. For example,

```
WHERE <expr> IN <subquery>
-- is equivalent to
WHERE <expr> = ANY <subquery>

-- and

WHERE <ex1> <op> ANY (
  SELECT <ex2> FROM <rel> WHERE <cond>
)
-- is equivalent to
WHERE EXISTS (
  SELECT * FROM <rel>
  WHERE <cond> AND <ex1> <op> <ex2>
)
```

Common SQL Constructs

Aggregate functions

Computes a single value from a set of tuples.

- Example: `SUM, AVG, MIN, MAX, COUNT`
- These functions are applied to all non-NULL values. `COUNT(*)` returns the number of rows, including `NULL` values.

Function	Input Type	Output Type
MIN	*any comparable type*	*same as input*
MAX	*any comparable type*	*same as input*
SUM	Numeric data *(e.g., INT, BIGINT, REAL, etc)*	SUM(INT) → BIGINT; SUM(REAL) → REAL
COUNT	*any data*	BIGINT

Grouping

Given `GROUP BY a1, ...` two tuple t and t' belong to the same group if $r1.a1 \equiv r2.a1, \ldots$

```
SELECT <expr>, <expr>, ...
FROM <relation>
[ WHERE <condition> ]
GROUP BY <expr>, <expr>, ...
[ HAVING <condition> ];
```

- Aggregation functions is done on each group.

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

`WHERE` is used to filter tuples before grouping, while `HAVING` is used to filter groups after grouping.

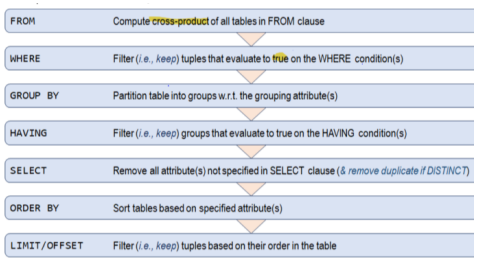
If column A_i of **table R** appears in the HAVING clause, either

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 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
```

Note: `ELSE` is optional. If omitted, the result is NULL if no condition is true.

COALESCE: Returns the first non-NULL value. Otherwise, return NULL.

```
-- returns 2
SELECT COALESCE(NULL, NULL, 2, 3);
-- returns NULL
SELECT COALESCE(NULL, NULL, NULL, NULL);
```

NULLIF: Returns NULL if $val_1 = val_2$, otherwise, returns val_1. Usually used with aggregation.

```
SELECT NULLIF(1, 1); -- returns NULL
SELECT AVG(NULLIF(price, 0));
```

Structuring SQL Queries

Common Table Expressions (CTE)

```
WITH
  <CTE_1> (col1, ..., coln) AS (<Q1>),
  <CTE_2> (col1, ..., coln) AS (<Q2>);
SELECT * FROM <CTE_1>;
```

Note:

1. Each CTEi is the name of a temporary table defined by query Qi
2. Each CTEi can reference any other CTEj that has been declared before CTEi (i.e., $j < i$)
3. The main SELECT statement Q0 can reference any possible subset of all CTEi

Views

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

```
CREATE VIEW <view_name>(col1, ..., coln)
     AS <query>;
DROP VIEW <view_name>;
```

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

Double negation: Transform the query to a form that is easier to work with (using only existential quantification!). For example:

```
Restaurants that sells all pizzas liked
by Holmer
```

Negating once

```
There exists a pizza liked by Holmer
that is not sold by any restaurant.
```

Negating twice

```
There does not exists a pizza liked
by Holmer but not sold by any restaurant.
```

Final query (using existential quantification)

```
-- the list of pizzas that holmer likes
-- but not sold by this restaurant
-- is empty
SELECT DISTINCT S1.rname FROM Sells S1
WHERE NOT EXISTS (
  -- list of pizzas liked by holmer
  -- but not sold by this restaurant
  SELECT 1 FROM Likes L
  WHERE L.cname = 'Homer'
  AND NOT EXISTS (
    SELECT 1 FROM Sells S
    WHERE S.pizza = L.pizza
    AND S.rname = S1.rname
  )
);
```

Cardinality: Given the previous prompt, derive two sets L and R where L refers to the set of pizzas sold by a single restaurant, and R refers to the set of pizzas liked by Holmer. A restaurant is said to satisfy the query if $R \subseteq L$. Now, we can iterate through the restaurants and compare their cardinality. If $|L| = |R \cup L|$, then the restaurant satisfies the query.

```
SELECT DISTINCT rname
FROM Sells S
WHERE (
  SELECT COUNT(DISTINCT pizza) FROM (
    SELECT pizza FROM Sells S1
    WHERE S1.rname = S.rname
    UNION
    SELECT pizza FROM Likes
    WHERE cname = 'Homer'
  ) AS T1
) = (
    SELECT COUNT(DISTINCT pizza)
    FROM Sells S1
    WHERE S1.rname = S.rname
);
```

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;
```

Note that Q0 is the ”base” case (where further queries are build on), and Q1 is the recursive case. Query is lazily evaluated and stops when a fixed point is reached.

SQL snippets

```
-- MAX
SELECT score FROM tbl TB1
WHERE NOT EXISTS (
  -- there does not exists a score
  -- that is higher than the current value
  SELECT 1 FROM tbl TB2
  WHERE TB1.score > TB2.score
);
```

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) $\rightarrow$ 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) $\rightarrow$ error if $A < B$	A	A
(total+key)-to-(key) $\rightarrow$ error if $A > B$	A	A
(total+key)-to-(total+key) $\rightarrow$ error if $A <> B$	A=B	A=B

ER guidelines

- If both entities has a total and key constraints, then merge them into one table with either A or B as PK. Otherwise, if only A has the total and key constraint, then merge the relation, R into A to form AR with A as the PK.
- If A has a key constraint and B has no upper-bound constraint, set A to be PK of the relation R.
- For ISA total constraints, create a view from the union of it's subclasses.

Queries Methods

- Set exception, union and intersection. Formulate the answer by identifying the appropriate sets.
- Double negation. Find set of all $entities, X$ that is not in the set of $entities, Y$ that failed $condition, c$.
- Scalar subqueries in the `SELECT` clause can be used to concat results.
- Add multiple entities in the `FROM` clause to cross join them.
- `WHEN-CASE` construction in `SELECT` clause

Notes

- Use a foreign key constraint for across tables validation.
- The division operator is useful to determine the set of entities in A that contains every value in B.
- If there is a `NULL` value in an `ALL` query, it will always return `FALSE`.
- If an entity contains exactly one of another entity, think about merging them into the same table.
- It will always return at least one row for the result, unless the aggregate function is used in conjunction with `GROUP BY`, in which case, there may not be a row, if there are no groups.

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.
SQL functions/procedures can be as complicated as needed (with control structures).
For example:
- `IF ... THEN ... END IF`
- `LOOP ... END LOOP`

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
```

Preferred:

```
CREATE OR REPLACE FUNCTION <NAME>(ARGS <TYPE
    >)
RETURNS TABLE(OUT_PARAMS TYPE) AS $$
:
$$ LANGUAGE sql
```

Examples:

```
CREATE OR REPLACE FUNCTION swap (INOUT val1
    INT, INOUT val2 INT)
RETURNS RECORD AS $$
DECLARE
  temp_val INTEGER;
BEGIN
  temp_val := val1;
  val1 := val2;  -- notice that output
       variables are explicitly assigned
  val2 := temp_val;
END;
$$ LANGUAGE plpgsql;
```

Procedures

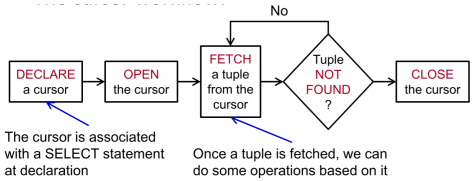
Procedures are reusable snippets that **do not return any value** i.e. update. They are transactional.
General syntax:

```
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 equvialent if their closures are equal. All candidate keys are functionally equivalent.

- $\alpha \to \beta$ is a **trivial** FD if $\beta \subset \alpha$.
- $\alpha \to \beta$ is a **non-trivial** FD if $\beta \not\subset \alpha$.
- $\alpha \to \beta$ 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

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

Main intuition: Suppose that B depends on a non-superkey $C_1C_2 \dots C_n$. Since $C_1C_2\dot{C}_n$ is not a superkey, the same $C_1C_2 \dots C_n$ may appear multiple times in the table. Whenever this happens, the same B would appear multiple times as well. This leads to redundancy.

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 $R_1 \cap R_2 \to R_1$ or $R_1 \cap R_2 \to R_2$
- 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.