- Relational Algebra
  - Structured Query Language (SQL) vs Relational Algebra (RA)
  - Operations
    - Basic
    - Set Operators
    - Derived
  - Basic Operators
    - Select Operator
      - Example
    - Project Operator
      - Example 1.
      - Example 2
  - Set Operators
    - Union Compatible Relations
      - Example
    - Union $\cup$
    - Intersect $\cap$
    - Set difference $-$
    - Cartesian Product $\bigtimes$
  - Derived Operators
    - Renaming
      - Example
    - Join
      - Example
    - Equijoin
      - Example
    - Natural Join
      - Example
    - Division
      - Example 1
      - Example 2
- Basic SQL
  - Partial Matching
    - Like Operator
  - Queries over multiple relations
  - Nested Queries/Subqueries
  - Subqueries and Complex Comparison Operators
    - IN Operator
      - IN operator used in subquery
    - ALL, ANY, and SOME used in Subqueries
      - Op ALL
      - Op SOME
      - Op ANY
        - Example 1
        - Example 2
    - EXIST
    - Division

        - Example 1
    - Aggregate Functions
        - Definition
        - Examples
    - GROUP BY and HAVING clauses
        - GROUP BY
            - Rules
        - HAVING clause
        - Exercises
- Null, Triggers, and Assertions
    - NULL Values
        - Issues
            - Comparison with NULL
            - Using Logical Operators
            - Impacts of SQL Constructs
        - More Integrity Constraints
            - NOT NULL
            - UNIQUE
            - CHECK
    - Assertions
    - Triggers

Relational Algebra

There are three variants for describing queries on a relational database

1. Relational algebra
2. Relational calculus
3. SQL
   Query languages are different to programming languages such that QLs are not expected to be *Turing complete*, but they support easy, efficient access to large data sets

Structured Query Language (SQL) vs Relational Algebra (RA)

1. SQL is predominant application-level query language while Relational Algebra is an intermediate language used within DBMS
2. SQL is **declarative** while RA is **procedural**

Operations

Basic

1. Selection (σ)
2. Project (π)

Set Operators

1. Cross Product ($\times$)
2. Set-difference ($-$)
3. Union ($\cup$)

Derived

1. Intersection

2. Join
3. Division
4. Renaming

Basic Operators

Select Operator

Select rows that satisfy the condition

$$\sigma_{condition}(relation)$$

Operators: $<, \leq, \geq, >, =, \neq$
Conditions

- `<attribute> operator <constant>`
- `<attribute> operator <attribute>`
- `<condition> AND <condition>`
- `<condition> OR <condition>`
- `NOT <condition>`

Example

id	*name*	*address*	*hobby*
1122	John	123-34 Ave	hockey
1133	Joe	125-34 Ave	biking
2232	Bob	7 Whyte Ave	hockey
5678	John	123-34 Ave	stamps

$\sigma_{hobby='hockey'}$

id	*name*	*address*	*hobby*
1122	John	123-34 Ave	hockey
2232	Bob	7 Whyte Ave	hockey

$$\sigma_{hobby='hockey'}(Person)$$

Project Operator

Project on (or pick a subset of columns)

$$\pi_{attribute\ list}(relation)$$

Example 1.

id	*name*	*address*	*hobby*
1122	John	123-34 Ave	hockey
1133	Joe	125-34 Ave	biking
2232	Bob	7 Whyte Ave	hockey
5678	John	123-34 Ave	stamps

Person

name	*hobby*
John	hockey
Joe	biking
Bob	hockey
John	stamps

$$\pi_{name,\ hobby}(Person)$$

Example 2

id	*name*	*address*	*hobby*
1122	John	123-34 Ave	hockey
1133	Joe	125-34 Ave	biking
2232	Bob	7 Whyte Ave	hockey
5678	John	123-34 Ave	stamps

id	*name*
1133	Joe
5678	John

$$\pi_{id,\ name}(\sigma_{hobby='hockey'\ OR\ hobby='hiking'}(Person))$$

Set Operators

Union Compatible Relations

Two relations are *union compatible* if:

1. Both have the same number of columns
2. Names of attributes are the same in both
3. Attributes with the same name in both relations have the same domain
   Union compatible relations can be combined using union, intersection, and set difference.

Example

Suppose there are tables

$$Person(SSN, Name, Address, Hobby)$$

and

$$Professor(Id, Name, Office, Phone)$$

. Tables `Person` and `Professor` are *NOT* union compatible but

$$\pi_{name}(Person)$$

and

$$\pi_{name}(Professor)$$

are union compatible. Which makes

$$\pi_{name}(Person) - \pi_{name}(Person)$$

possible.

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

Union $\cup$

Generates a relation (set - no duplicates) combining all the entries in both table

Intersect $\cap$

Generates a relation (set) of entries that are common in both tables

Set difference $-$

Generates a relation that contains only the tuples that appear in the first and not the second of the input relations

Cartesian Product $\times$

$R \times S = \{< x, y > | x \in R, y \in S\}$

- $R \times S$ is the set of all concatenated tuples in $< x, y >$
- Relations **DO NOT** have to be union-compatible

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- Each row of S1 is paired with each row of R1.
- Result schema has one field per field of S1 and R1
- *Conflict*: Both S1 and R1 have a field called sid.

Derived Operators

Renaming

$\rho_{x(A1,A2,...,An)} expr$

- where `expr` returns a relation
- Rename the resulting relation to `x` with the first column in the result relating to `A1`, the second to `A2`, etc.
- Used to clean up the result, and prepare it for the next operation

Example

Consider tables

$$Transcript(sid, cid, sem, grade)$$

$$Teaching(pid, cid, sem)$$

$Transcipt \times Teaching$ is not defined but the following is:

$$\rho_{sid,cid1}(\pi_{sid,cid}(Transcript)) \times \rho_{pid,cid2}(\pi_{pid,cid}(Teaching))$$

Join

$$R \bowtie_{\text{join-condition}} S$$

Where join-condition is a conjunction of terms $A_i \; oper \; B_i$ in which

- A_i is an attribute of R
- B_i is an attribute of S
- $oper$ is one of $<, \leq, \geq, >, =, \neq$
  This is equivalent to $\sigma_{\text{join-condition}'}(R \times S)$

Example

Find employees who earn more than their managers. Consider the tables

$$\text{Employee(name, id, salary, mngrId)}$$

$$\text{Manager(name, id, salary)}$$

The following join will

$$\pi_{\text{Employee.name}} (\text{Employee} \bowtie_{\text{mngrId=manager.id AND Employee.salary} > \text{Manager.salary}} \text{Manager})$$

Note that the join will yield a table with attributes `Employee.name, Employee.id, Employee.salary, Employee.mngrId, Manager.name, Manager.id, Manager.salary`

Equijoin

A special case of Join where all the conditions are equalities. This results in a schema similar to cross-product but only one copy of fields for which equality is specified

Example

$$\pi_{\text{name, cid}} (\text{Student} \bowtie_{\text{id=sid}} \sigma_{\text{grade='A'}}(\text{Transcript}))$$

Student

id	*name*	*addr*	*status*
111	John	.....	.....
222	Mary	.....	.....
333	Bill	.....	.....
444	Joe	.....	.....

Transcript

sid	*cid*	*sem*	*grade*
111	114	F10	B
222	115	F10	A
333	201	W10	A

Mary	115
Bill	201

The equijoin is very useful since it combines related data in different relations.

Natural Join

A special case of join where it equates **all and only those attributes with the same name**. Duplicate columns eliminated from the result

Example

Suppose you have tables

$$\text{Transcript}(\text{sid, cid, sem, grade})$$

$$\text{Teaching}(\text{pid, sid, sem})$$

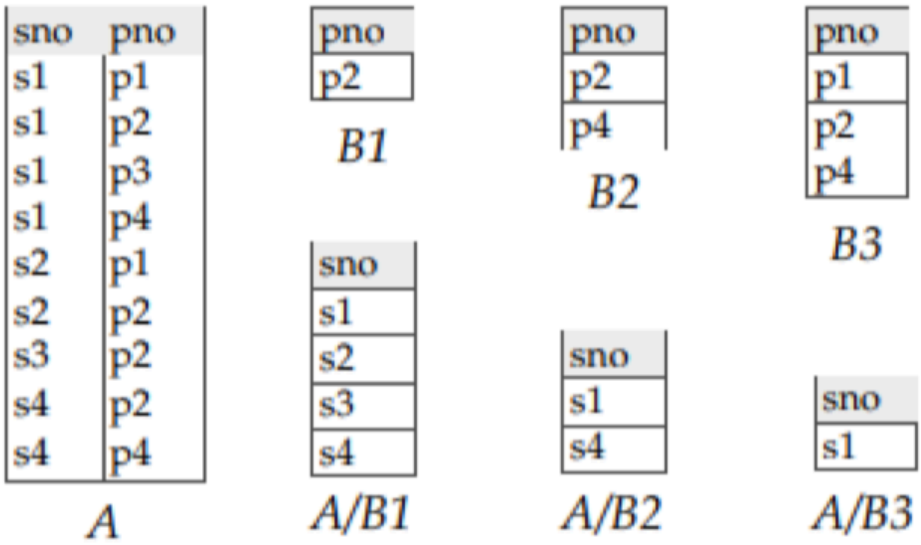
$$\text{Transcript} \bowtie \text{Teaching} = \pi_{\text{sid, Transcript.cid, Transcript.sem, grade, pid}} (\text{Transcript} \bowtie_{\text{Transcript.cid=Teaching.cid AND Transcript.sem=Teaching.sem}} \text{Teaching})$$

Division

Find tuples in on relation r, that matches all tuples in another relation s. This can be expressed in terms of project, set difference, and cross-product

- $r(A_1, \ldots, A_n, B_1, \ldots, B_m)$
- $s(B_1, \ldots B_m)$
- r/s with attributes $A_1, \ldots, A_n$ is the set of all tuples $< a >$ such that for every tuple $< b >$ in s, $< a, b >$ is in r

Example 1



Example 2

List the Ids of students who have passed all courses taught in winter 2010

Numerator

$$\pi_{\text{sid, cid}}(\sigma_{\text{Grade}\neq\text{'F'}}(\text{Transcript}))$$

Denominator

$$\pi_{\text{cid}}(\sigma_{\text{Sem}=\text{'W10'}}(\text{Teaching}))$$

Then the result is $\text{Numerator}/\text{Denominator}$

Basic SQL

Partial Matching

```
SELECT attr
FROM tbl
WHERE attr LIKE '<partial_matching>'
```

Like Operator

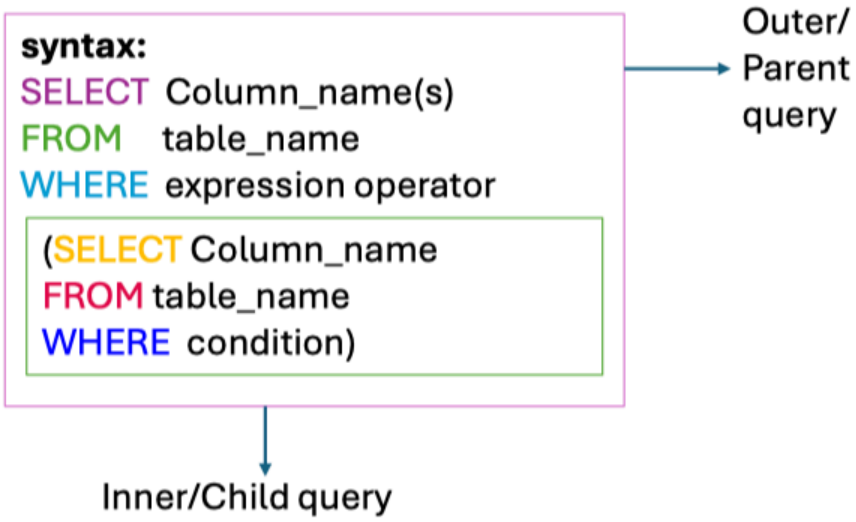
- `%` (percent): Matches any sequence of characters (including zero characters)
  - `A%` - match any sequence that starts with latter `A`
  - `%son` - match sequence that ends with the `son`
- `_` (underscore): Matches exactly one character
  - `A__e` - match with values that starts with `A`, has any two characters in the middle and ends with `e`
  - `g__%` - match values that start with `g` and are at least 3 characters in length

Queries over multiple relations

```
SELECT branch.bname, assets
FROM branch, customer, deposit
WHERE customer.city = 'Jasper'
        AND customer.cname = deposit.cname
        AND deposit.bname = branch.bname
```

Nested Queries/Subqueries

A subquery is a query within a query. The inner query is executed first, afterwards the results is passed to the outer query



Subqueries and Complex Comparison Operators

IN Operator

Used to specify multiple possible values for a column

```
SELECT
FROM
WHERE expr | (expr list) IN (set of values)
```

IN operator used in subquery

Find every customer who has a deposit in some branch at which John Doe has a deposit

```
# deposit(accno(pk), cname, bname, balance)
SELECT cname
FROM deposit
WHERE bname IN
        (SELECT bname
        FROM deposit
        WHERE cname = 'John Doe')
```

However we can use the same example without Subquery

```
SELECT d1.cname
FROM deposit d1, deposit d2
WHERE d2.cname = 'John Doe' AND d1.bname = d2.bname
```

ALL, ANY, and SOME used in Subqueries

```
SELECT Column_name(s)
FROM table_name
WHERE expression_operator < ALL
(SELECT Column_name
 FROM table_name
 WHERE column DATA TYPE(value1, value2, value3))
```

Op ALL

(set of values) evaluates to true iff the comparison evaluates to true for every value in the set

- `<=ALL`, `=ALL`, `>=ALL`, `>ALL`, `<ALL`

Op SOME

(set of values) evaluates to true iff the comparison evaluates to true for at least one value in the set

- `<=SOME`, `=SOME`, `>=SOME`, `>SOME`, `<SOME`

Op ANY

(set of values) evaluates to true iff the comparison evaluates to true for any value in the set

- `<=ANY`, `=ANY`, `>=ANY`, `>ANY`, `<ANY`

Example 1

Find branches that have assets greater than the assets of all branches in Calgary

```
# branch(bname(pk), address, city, assets)
SELECT bname
FROM branch
WHERE assets > ALL (
        SELECT assets
        FROM branch
        WHERE city = 'Calgary')
```

Example 2

Identify employees earning annual salary that's less than or equal to the annual salaries earned by all employees in the following roles - Manager, Head chef, Assistant Manager, Head Waiter

```
SELECT * FROM Employees
WHERE Annual_Salary <= ALL (
        SELECT Annual_Salary
        FROM Employees
        WHERE Role IN ('Manager', 'Head Chef', 'Assist. Manager', 'Head waiter'))
```

EXIST

- `EXISTS(SELECT ...)` evaluates to true iff the result of the subquery contains at least one row
- The expression is evaluated for every iteration of the outer query

```
# branch(bname(pk), address, city, assets)
# customer(cname(pk), street, city)

# Example 1: Find the names of customers who live in a city with no bank branches
SELECT cname
FROM customer
WHERE NOT EXISTS (
        SELECT *
        FROM branch
        WHERE customer.city = branch.city
)

# Example 2: Find the names of customers who live in a city which has a bank branch
SELECT cname
FROM customer
WHERE EXISTS (
        SELECT *
        FROM branch
```

```
        WHERE customer.city = branch.city
)
```

Division

Find the subset of items in one set that are related to all items in another set

Example 1

Find customers who have deposit accounts in all branches. Strategy is

1. Find set *A*, of all branches in which a particular customer *c* has a deposit account
2. Find set *B* all branches
3. Output *c* of $A \supset B$ or equivalently, if *B-A* is empty
4. **Relation A contains relation B == NOT EXISTS (B Except A)**

```
# branch(bname(pk), address, city, assets)
# customer(cname(pk), street, city)
SELECT c.cname
FROM customer c
WHERE NOT EXISTS
        (SELECT b.bname --set B of all branches
        FROM branch b
        EXCEPT
        SELECT d.bname -- set A of branches in which custmer c has a deposit account
        FROM deposit d
        WHERE d.cname=c.cname)
```

Aggregate Functions

Aggregate are functions that take a collection of values as input and return a single value. Note that if the SELECT clause uses an aggregate operation, then it must use only aggregate operations unless the query contains a GROUP BY clause

Definition

1. `COUNT([DISTINCT] A)`: The number of (unique) values in the A column
    - Not allowed to use `DISTINCT` with `COUNT(*)`, must include an attribute in the `COUNT`
    - If `DISTINCT` is not used, then `COUNT(*) = COUNT(x)`
2. `SUM([DISTINCT] A)`: The sum of all (unique) values in the A column
3. `AVG([DISTINCT] A)`: The average (unique) values in the A column
4. `MAX(A)`: The maximum value in the A column
5. `MIN(A)`: The minimum value in the A column

Examples

branch (bname, address, city, assets)
customer(cname, street, city)
deposit(accno, cname, bname, balance)
loan(accno, cname, bname, amount)

1. Find the number of customers Edmonton

```
SELECT COUNT(*)
FROM customer
WHERE city = 'Edmonton'
```

2. Find the total assets of branches in Edmonton

```
SELECT SUM(assets)
FROM branch
WHERE city='Edmonton'
```

3. Find the number of different branches where John Doe has a deposit account

```
SELECT COUNT DISTINCT(bname)
FROM deposit
WHERE cname = 'John Doe'
```

4. Find the name of the customer with the highest loan amount

```
SELECT L.cname, L.amount
FROM loan L
WHERE L.amount = (SELECT MAX(L2.amount) FROM loan L2)
```

5. Find the number of customers who have deposit accounts in at least 3 different branches

```
SELECT COUNT(*)
FROM customer c
WHERE 3 <= (SELECT COUNT(DISTINCT d.bname)
                        FROM deposit d
                        WHERE c.cname = d.cname)
```

6. Find the name of branches which have assets greater than the average assets of all branches

```
SELECT b.bname
FROM branch b
WHERE b.assets > (SELECT AVG(b1.assets)
                                    FROM branch b1)
```

GROUP BY and HAVING clauses

GROUP BY

```
SELECT column_name1, column_name2
FROM table_name
GROUP BY column_name1, column_name2

SELECT column_name1, column_name2
FROM table_name
WHERE filter_condition
GROUP BY column_name1, column_name2
```

Rules

1. The columns after the GROUP BY clause must be used for the groupings
2. If there is a WHERE clause in the SELECT statement, then the GROUP BY clause must be placed after the WHERE clause
3. Every column name in the SELECT clause must appear in the GROUP BY clause

HAVING clause

```
SELECT column_name1, column_name2
FROM table_name
WHERE filter_condition
GROUP BY column_name1, column_name2
HAVING group_filter_condition
```

- The HAVING clause is used to specify a filter condition for the group generated by the GROUP BY clause

- The WHERE clause cannot be used to filter grouped data so we use HAVING

Exercises

branch (bname, address, city, assets)
customer(cname, street, city)
deposit(accno, cname, bname, balance)
loan(accno, cname, bname, amount)

1. Find the cities more than two bank branches

```
SELECT city
FROM branch
GROUP BY city
HAVING COUNT(*) > 2;
```

2.. For each customer having more than two deposit accounts, find the name and the city of the customer

```
SELECT c.cname, c.city
FROM customer c, deposit d
WHERE c.cname = d.cname
GROUP BY c.cname, c.city
HAVING COUNT(*) > 2
```

3. Find branches with an average account balance greater than $1200

```
SELECT d.bname
FROM deposit d
GROUP BY d.bname
HAVING AVG(d.balance) > 1200
```

4. Find branches with an average account balance greater than or equal to the average account balances of all branches

```
SELECT d.bname
FROM deposit d
GROUP BY d.bname
HAVING AVG (d.balance) >= (SELECT AVG(d1. balance) FROM deposit d1)
```

5. Find cities of all customers who have total deposit balances of over $4000

```
SELECT c.city
FROM customer c, deposit d
WHERE c.cname, d.cname
GROUP BY c.city
HAVING SUM(d.balance) > 4000
```

Null, Triggers, and Assertions

NULL Values

SQL uses NULL values to represent unknown/inapplicable values

Issues

Comparison with NULL

`IS NULL` - if a column value is NULL, evaluates to `True` otherwise `False`

Using Logical Operators

Raises an important point suggesting that the logical operators (AND, OR, NOT) must be defined with 3-valued outcomes:

1. True
2. False
3. Unknown

Impacts of SQL Constructs

1. WHERE clause eliminates rows which do not evaluate to True
2. The presence of NULL values in any row that evaluates to False or Unknown is eliminated (*count False only*)
3. Significantly impacts the results in queries especially subqueries involving EXISTS
4. Aggregate operators has unexpected behaviour dealing with NULL values
5. `COUNT(*)` handles NULL values like other values and count them among all values
6. `SUM`, `AVG`, `MIN`, `MAX` simply discard NULL values
7. NULL can be disallowed by specifying NOT NULL

More Integrity Constraints

NOT NULL

A domain constraint the **prohibits the insertion of null value**

```
name varchar(20) NOT NULL
```

UNIQUE

Ensures that each value in a column or set of columns is unique across all rows in the table

- A table can have more than one unique values unlike primary keys
- Unique allows for NULL values unlike primary keys

```
name varchar(20) UNIQUE
```

CHECK

A common use of the check clause is to ensure that attribute values satisfy specified conditions. Checked every time a tuple is inserted or updated. Violations are rejected

```
CREATE TABLE branch
        (bname CHAR(15) NOT NULL,
        address VARCHAR(20),
        city CHAR(9),
        assets DECIMAL(10, 2) DEFAULT 0.00,
        CHECK (assets >= 0));
```

```
CREATE TABLE deposit
        (accno CHAR(9) NOT NULL,
        cname VARCHAR(15),
        bname VARCHAR(15),
        balance DECIMAL (10, 2) DEFAULT 0.00,
        CHECK (cname='Bill Clinton' OR balance > 100000))
```

Assertions

For when we want to set constraints over all tables (global constraints). Checked before any insertion or update can be committed to the database perform the check and apply the specified condition

```
CREATE ASSERTION name CHECK (condition)
```

```
CREATE ASSERTION deposit CHECK
        (NOT EXISTS
                (SELECT bname, cname
                FROM loan
                WHERE amount > 100000
                GROUP BY bname, cname
                HAVING COUNT (*) > 2));
```

Triggers

A set of actions in the form of stored program. These set of actions are invoked automatically when certain actions occur. Each trigger name must be **unique**

```
CREATE TRIGGER unique_trigger_name
        [BEFORE | AFTER | INSTEAD OF]
        [INSERT | UPDATE | DELETE]
        ON table_name
        [WHEN CONDITION]
BEGIN
        statements;
END;
```

Example: Log the old address information into an address_log table whenever there is a change in the address

```
# customer(cname(pk), street, city)
# addresslog(cname(pk), ctime, ostreet, ocity, nstreet, ncity)

CREATE TRIGGER log-addresses
        AFTER UPDATE ON customer
        WHEN old.street <> new.street OR old.city <> new.city
BEGIN
        INSERT INTO addresslog VALUES
                (old.cname, datetime('now'), old.street, old.city, new.street, new.city);
END;
```