Chapter 3

SQL
Relational Commercial Languages

- **SQL (Structured Query Language)**
  - the standard relational DB language
  - used *interactively* which transforms inputs into desired outputs via relations
  - relations are closed under the SQL operations

- **Data Definition Language (DDL) of SQL**
  - create/delete/modify relation schemas/views/indices
  - **Create table**: specifies a new relation schema
    
    ```
    create table r(A_1 D_1, ..., A_n D_n)
    ```
    
    where
    - *r* is new relation name
    - *A_i*, \(1 \leq i \leq n\), an attribute name
    - *D_i*, \(1 \leq i \leq n\), domain of *A_i*
Relational Commercial Languages

- **Data Definition Language (DDL) of SQL**
  - *Drop table*: removes an existing relation
    - `drop table r`: deletes all information of `r` (schema + tuples in `r`)
    - `delete r`: deletes all tuples in `r` but retains the schema of `r`
  - *Alter table*: adds an attribute to an existing relation
    
    ```
    alter table r add A D
    ```
    ```
    alter table r drop A
    ```
    
    where
    
    `r` is an existing relation
    `A` is an attribute name
    `D` is the domain of `A`

    Note: the new attribute will have NULLs in all tuples of `r`
Relational Commercial Languages

- **Data Manipulation Language** (DML) of SQL
  - a query language based on the relational algebra and tuple calculus
  - retrieve/insert/delete/modify tuples in DB relations

- **DML operations:**
  - Select
    - format: `SELECT A_1, A_2, ..., A_n`
    - Usage: projects attributes $A_1, A_2, ..., A_n$ in order
      - *duplicate* rows remain
      - use the qualifier, `DISTINCT`, to remove duplicate rows
SQL

- DML operations:
  - **Where**
    - format: `WHERE P`
    - Usage: selects tuples which satisfy the predicate `P`
  - **From**
    - format: `FROM r_1, r_2, ..., r_m`
    - Usage: performs a cartesian product on relations `r_1, r_2, ..., r_m`

- Typical SQL Query:
  
  ```sql
  SELECT A_1, A_2, ..., A_n
  FROM r_1, r_2, ..., r_m
  WHERE P
  ```

  - in Relational Algebra: \( \pi_{A_1, A_2, ..., A_n} (\sigma_P (r_1 \times r_2 \times ... \times r_m)) \)
  - *Where* clause is *optional* and `P` involves attrs appear in the *From* clause
  - `A_1, A_2, ..., A_n` can be replaced by ‘*’ to select all attributes in the query
SQL

Set (union/intersect/subtraction) operations:

- Combine the results of two SELECT statements into a single one
- Apply only to union compatible relations
  - **Union** (\(\cup\)): returns those tuples generated by either query and eliminates duplicate tuples (use **Union all** to retain duplicates)
  - **Intersect** (\(\cap\)): returns only those tuples generated by both queries
  - **Minus** (\(-\)): returns only those tuples generated by the first query but not the second query

Format:

```
(SELECT …) UNION/ INTERSECT/ MINUS (SELECT …)
```

E.g., Let is_taking(Sname, Course#) and has_taken (Sname, Course#) be 2 relations

```
(SELECT DISTINCT Course# 
  FROM is_taking 
  WHERE Sname = ‘Joe Young’)
UNION
(SELECT DISTINCT Course# 
  FROM has_taken 
  WHERE Sname = ‘Joe Young’)
```
Set membership:

- **Usage:** IN/NOT IN is used as part of the WHERE-clause to indicate a *set membership* or *absence* of set membership.

- **Retrieval/update using subquery:** IN (NOT IN)

  Format:

  ```sql
  SELECT A_1, A_2, ..., A_n
  FROM r_1, r_2, ..., r_m
  WHERE P (NOT) IN subquery
  ```

- **NOTE:** works from *bottom-up*, i.e., a subquery is evaluated first prior to the evaluation of the main (outer) query.

**Example:** Given `Is_taking(Sname, Course#)` and `Has_taken(Sname, Course#)`

```sql
SELECT DISTINCT Sname
FROM Is_taking I
WHERE Course# [NOT] IN
  (SELECT Course#
   FROM Has_taken H
   WHERE H.Sname = I.Sname)
```
SQL

- **Set Comparison:**
  - Tests a single value against elements of a set
  - Format: `SELECT ... WHERE A_i Comp subquery`
    
    where Comp ∈ {< some, > some, ≤ some, ≥ some, = some, ≠ some, < all, > all, ≤ all, ≥ all, = all, ≠ all }

- **Example.** Given the relation `Student(SSNO, Sname, Age)`

  ```sql
  SELECT SSNO
  FROM Student
  WHERE Age > Some
  (SELECT DISTINCT Age
   FROM Student
   WHERE Sname = 'Joe Young')
  ```
Definition of “Some” Clause in SQL

- $E \prec \text{some} \succ r \iff \exists t \in r$ such that $(E \prec \text{some} t)$, where $\prec \in \{<, \leq, >, \geq, =, \neq\}$

\[
\begin{array}{c|c|c|c}
\text{5} & \text{some} & \text{0} & \text{true} \\
\text{5} & \text{some} & \text{5} & \text{false} \\
\text{5} & \text{some} & \text{6} & \\
\end{array}
\]

(5 < some 5) = true (read: 5 < some tuple in the relation)

(5 < some 5) = false

(5 = some 5) = true

(5 \neq some 5) = true (since 0 \neq 5)

Note that “= some” \equiv IN, but “\neq some” \neq NOT IN
Definition of “All” Clause in SQL

- \( E <\text{comp}> \text{all} \ r \iff \forall t \in r \ (E <\text{comp}> \text{all} t) \)

\[
\begin{array}{c|c|c}
\text{5} & \text{0} & \text{false} \\
\text{6} & & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{6} & \text{5} & \text{false} \\
\text{10} & & \text{true} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{4} & \text{5} & \text{false} \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{4} & \text{5} & \text{false} \\
\text{6} & & \text{true} (\text{since } 5 \neq 4 \text{ and } 5 \neq 6) \\
\end{array}
\]

Note that “\(\neq \text{all}\)” \(\equiv\) NOT IN, but “\(= \text{all}\)” \(\neq\) IN
(Sub)Set Comparison:

- compares sets (not) being subsets of other sets
  
  - Format: \texttt{SELECT … WHERE subquery CONTAINS/NOT CONTAINS subquery}

Example. Given Is\_taking(\text{Sname, Course#}) and Has\_taken(\text{Sname, Course#})

\begin{verbatim}
SELECT I.Sname
FROM Is\_taking I
WHERE (SELECT DISTINCT H.Course#
    FROM Has\_taken H
    WHERE I.Sname = H.Sname)
[NOT] CONTAINS
    (SELECT DISTINCT IT.Course#
        FROM Is\_taking IT
        WHERE IT.Sname = I.Sname)
\end{verbatim}
Set Comparison:

- Tests for (non-)empty relations generated by subqueries
  - Format: SELECT … WHERE EXISTS/NOT EXISTS subquery

Example. Consider the following database schema:

customer(customer-name, street, customer-city)
deposit(branch-name, account-number, customer-name, balance)
borrow(branch-name, loan-number, customer-name, amount)

Query: “Find all customers who have both an account and a loan at the Perryridge branch”

```
SELECT customer-name
FROM customer C
WHERE EXISTS (SELECT *
               FROM deposit D
               WHERE D.customer-name = C.customer-name AND branch-name = ‘Perryridge’)
AND EXISTS (SELECT *
             FROM borrow B
             WHERE B.customer-name = C.customer-name AND branch-name = ‘Perryridge’)
```
SQL

- Joins
  - defined in terms of $\times$ (FROM), $\sigma$(WHERE) and $\pi$(SELECT)
  
  - Format: Same as a typical SQL query except
    1. Attributes appear in more than one relation should be designated with relation name (tuple variable)
    2. WHERE-clause specifies the join on common attributes from different relations

- **Example.** Given *Borrower(customer-name, loan-number)* and *Loan(loan-number, branch-name, amount)*

  ```sql
  SELECT customer-name, T.loan-number, amount
  FROM Borrower T, Loan S
  WHERE T.loan-number = S.loan-number
  ```
SQL

- The Except Operation:
  - Same as set minus (difference)
  - Automatically eliminate duplicates. To retain all duplicates, use except all
  - Format: Subquery Except Subquery
  - Example. Given the following relations
    Depositor(account-number, customer-name, balance) and Borrower(loan-number, customer-name, amount)
    Find all customers who have an account but no loan.
    
    (SELECT customer-name FROM depositor) EXCEPT (SELECT customer-name FROM borrower)
The Rename Operation

- SQL allows relations and attributes to be renamed using the AS clause:

  \[ \text{old-name AS new-name} \]

- **Example.** “Find the name, loan number, and loan amount of all customers; rename \textit{loan-number} as \textit{loan-id}.”

  /* Given
  Borrower (Customer-name, Loan-number)
  Loan (Loan-number, Branch-name, Amount) */

  SELECT customer-name,
         \( B.\text{loan-number} \text{ AS loan-id,} \)
         amount
  FROM  borrower \( B \), loan \( L \)
  WHERE \( B.\text{loan-number} = L.\text{loan-number} \)
Derived Relations

- Using subqueries in the FROM clause to generate derived relations.
- Data in any derived relations can be used in the WHERE clause of the same SQL statement.
- **Example.** “Find the **cheapest** laptops (identified by Model) that have the **largest** hard drive among all the laptops in the DB,” where

```sql
/* Given Laptop(Model, Speed, RamSize, HdSize, Screen, Price) */

SELECT Model
FROM (SELECT Model, Price
       FROM Laptop
       WHERE HdSize >= ALL (SELECT L.HdSize FROM Laptop L))
WHERE Price <= ALL
  (SELECT Price
   FROM Laptop
   WHERE HdSize >= ALL (SELECT L.HdSize FROM Laptop L))
```
Derived Relations

- A derived relation cannot be used in another subquery of the same SQL statement

- Example. “Find the cheapest laptops that have the largest hard drive among all the laptops in the DB.”

```sql
SELECT Model
FROM (SELECT Model, Price
      FROM Laptop
      WHERE HdSize >= ALL (SELECT L.HdSize FROM Laptop L))
      MAXSize
WHERE Price <= ALL
      (SELECT M.Price
       FROM MaxSize M);

/* An invalid Oracle SQL Query due to the scope error */
Derived Relations

- Each derived relation can be assigned a name and can be used in the same SQL statement.

- Example. “Find the laptops that have the largest hard drive among all the laptops in the database and cost less than $2,500.”

```sql
SELECT Model
FROM (SELECT Model, Price
FROM Laptop
WHERE HdSize ≥ ALL
    (SELECT L.HdSize FROM Laptop L)) MaxSize
WHERE MaxSize.Price < 2500;
```
Database Updates in SQL

- **Deletion**: removes whole tuples
  - Format: `DELETE r [WHERE P]`

- **Insertion**: adds new tuples with correct arity and domain values
  - Format: `INSERT INTO r[A_1, A_2,…, A_n] VALUES (V_1, V_2,…, V_n)`
    - Where $V_i \in \text{dom}(A_i)$, $1 \leq i \leq n$. If $A_1, A_2,…, A_n$ are not specified, then $V_1, V_2,…, V_n$ must be listed according to the order of attributes shown in the schema.

- **Modification**: modifies tuple values in a relation $r$
  - Format: `Update r SET attr = value [WHERE P]`
Modification of the Database – Insertion

- **Example.** Given Account(Account-number, Branch-name, Balance), add a new tuple to `account`

  ```sql
  INSERT INTO account
  VALUES ('A-9732', 'Perryridge', 1200)
  ```

  or equivalently

  ```sql
  INSERT INTO account (branch-name, balance, account-number)
  VALUES ('Perryridge', 1200, 'A-9732')
  ```

- **Example.** Add a new tuple to `account` with `balance` set to `null`

  ```sql
  INSERT INTO account
  VALUES ('A-777', 'Perryridge', null)
  ```
Modification of the Database – Insertion

- A Select-From-Where statement should be fully evaluated before any of its results are inserted into the relation; otherwise, the query

  \[
  \text{INSERT INTO } \text{table1 SELECT * FROM table1}
  \]

would cause problems.

- Example. “Provide as a gift for all loan customers of the Perryridge branch, a $200 savings account. Let the loan number serve as the account number for the new savings account.”

  Account(Acct-number, Branch-name, Balance)
  Loan (Loan-number, Branch-name, Amount)

  \[
  \text{INSERT INTO account SELECT loan-number, branch-name, 200 FROM loan WHERE branch-name = 'Perryridge'}
  \]
  \[
  \text{/* Depositor(Customer-name, Acct-number) */}
  \]

  \[
  \text{INSERT INTO depositor SELECT customer-name, L.loan-number FROM loan L, borrower B WHERE branch-name = 'Perryridge'} AND \]
  \[
  L.loan-number = B.loan-number
  \]
Modification of the Database – Updates

- **Example.** “Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.”

  - Write two `update` statements:

    ```sql
    UPDATE account
    SET balance = balance * 1.06
    WHERE balance > 10000
    
    UPDATE account
    SET balance = balance * 1.05
    WHERE balance <= 10000
    ```

  - The order is important
  - Can be done better using the `case` statement
“Case” Statements for DB Updates

- **Example.** “Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%” using a **case** statement.

```sql
UPDATE account
SET balance = CASE
    WHEN balance <= 10000 THEN balance * 1.05
    ELSE balance * 1.06
END
```
“Case” Statements for DB Retrieval

Example. “Find the name, (salary) category, and department of each employee, sorted by category and department. The salary categories include high (i.e., $100,000 \leq \text{Salary}$), medium (i.e., $50,000 \leq \text{Salary} \leq 100,000$) and low (i.e., $\text{Salary} \leq 50,000$).

```sql
/* Employee(Ename, ESSN, Address, Salary);
   Works_for(ESSN, Dno) */

SELECT Ename, Category, DNO
FROM Employee E, Works_for W,
     (SELECT ESSN AS CSSN,
      CASE
       WHEN Salary >= 100000 THEN 'High'
       WHEN Salary < 100000 AND Salary >= 50000 THEN 'Medium'
       ELSE 'Low'
      END AS Category
     FROM Employee)
WHERE E.ESSN = CSSN AND E.ESSN = W.ESSN
ORDER BY Category, Dno;
```
“Case” Statements for DB Retrieval

Example. “Find the crew members (identified by $CID$) who have been or have not been assigned to at least one flight. (The result table must include two columns, $CID$ and Assigned, such that the column values of Assigned are either `Yes', for assigned, or `No', for not assigned.)

```sql
/* Given crew (CID, Name, Position, Fly_Hours) assign (Flight#, CID) */

SELECT CID, (CASE WHEN CID IN (SELECT CID FROM Assign) THEN 'Yes' ELSE 'No' END) AS Assigned
FROM Crew;
```
Modification of the DB – Deletion

- **Example.** Delete all account records at the *Perryridge* branch

  ```sql
  DELETE FROM account
  WHERE branch-name = 'Perryridge'
  ```

- **Example.** Delete all accounts at every branch located in *Needham*

  ```sql
  Account (Account-number, Branch-name, Balance)
  Branch (Branch-name, Branch-city, Assets)
  Depositor (Customer-name, Account-number)
  
  DELETE FROM depositor
  WHERE account-number IN
  (SELECT account-number
   FROM branch B, account A
   WHERE branch-city = 'Needham' AND
   B.branch-name = A.branch-name)
  
  DELETE FROM account
  WHERE branch-name IN
  (SELECT branch-name
   FROM branch
   WHERE branch-city = 'Needham')
  ```

  Must be in one transaction; otherwise, the order must be specified as is.
Views

- Provide a mechanism to hide certain data from the view of certain users.

- To create a view we use the command:

  \[
  \text{CREATE VIEW } \nu \text{ AS } \text{<query expression>}
  \]

  where

  - \text{<query expression>} is any legal expression
  - The (new) \textit{view name} is represented by \nu
Example Queries of Views

- A view consisting of the branches and all their customer names

```
CREATE VIEW all-customer AS
(SELECT branch-name, customer-name
 FROM depositor D, account A
 WHERE D.account-number = A.account-number)
UNION
(SELECT branch-name, customer-name
 FROM borrower B, loan L
 WHERE B.loan-number = L.loan-number)
```

- Find all the customers of the Perryridge branch

```
SELECT customer-name
FROM all-customer
WHERE branch-name = 'Perryridge'
```
SQL

- **Aggregate Functions:**
  - functions: `avg`, `count`, `sum`, `max`, `min` apply to a list of values appearing in the aggregated column(s) to yield a single value
  - functions on groups of tuples (generated using `Group By` clause)
    - Format:
      ```sql
      SELECT aggregate function(A_1, A_2, ..., A_n)
      FROM r_1, r_2, ..., r_m
      WHERE P
      GROUP BY (B_1, ..., B_k) /* GROUP BY is optional */
      ```
    - Usage: group together tuples that have the *same* value of specified attribute(s) expressed in the `GROUP BY` clause
    - **Constraint:** in the `SELECT` clause that includes aggregations, only those attributes mentioned in the `GROUP BY` may appear unaggregated
  - With *aggregate functions*, SQL is strictly more powerful than the *relational algebra*, but is less powerful than a general-purpose *programming language*
SQL Aggregate Functions

- Perform some *computation/summarization*
- A significant extension of *relational algebra*
- SQL supports five aggregate operations:
  1. **COUNT** ([DISTINCT] A): *Number* of (unique) values in A
  2. **SUM** ([DISTINCT] A): *The sum* of all (unique) values in A
  3. **AVG** ([DISTINCT] A): *The average* of all (unique) values in A
  4. **MAX** (A): *The maximum* value in A
  5. **MIN** (A): *The minimum* value in A

where A is a column (i.e., an attribute), and

SUM and AVG operate only on a set of numerical values
Examples of SQL Aggregate Functions

- Let crew(ID#, Name, Position, Seniority, Fly_Hours) be a relation schema

- Find the *average flying hours* of all crew members.
  
  \[
  \text{SELECT AVG(Fly\_Hours) FROM crew;}
  \]

- Find the *average flying hours* of all crew members who are *pilots*.
  
  \[
  \text{SELECT AVG(Fly\_Hours) FROM crew WHERE Position = 'Pilot';}
  \]

- Find the *name* and *flying hours* of the crew member who has accumulated the *most* flying hours.
  
  \[
  \text{SELECT Name, MAX(Fly\_Hours) FROM crew;}
  \]

NOTE: illegal query in SQL.
Examples of SQL Aggregate Functions

- **Example.** Given the following relations:
  
  crew(ID#, Name, Position, Seniority, Fly_Hours)
  flight(Flight#, Source_City, Dest_City, Date, No_Pass, Dep_Time, Arr_Time)

- Find the *name* and *flying hours* of the crew member who has accumulated the most flying hours.
  
  ```sql
  SELECT Name, Fly_Hours
  FROM crew
  WHERE Fly_Hours = (SELECT MAX(Fly_Hours)
                       FROM crew)
  ```

- Find the number of *source cities* in the database.
  
  ```sql
  SELECT COUNT(DISTINCT Source_City) AS Num_Source_City
  FROM flight;
  ```
SQL Aggregate Functions with Group-By

Example. Given the following relations:

crew(ID#, Name, Position, Seniority, Fly_Hours)
flight(Flight#, Source_City, Dest_City, Date, No_Pass, Dep_Time, Arr_Time)

Find the lowest seniority for each position.

```sql
SELECT Position, MIN(Seniority)
FROM crew
GROUP BY Position;
```

Find the number of flights originated from a particular city.

```sql
SELECT Source_City, COUNT(Dep_Time)
FROM flight
GROUP BY Source_City;
```

Find the latest flight (identified by dep_time) before 8:00 p.m. from each city.

```sql
SELECT Source_City, MAX(Dep_Time)
FROM flight
WHERE Dep_Time < 2000
GROUP BY Source_City;
```
SQL Aggregate Functions with Group-By

- **Example.** Given the following relations:
  
  - `crew(ID#, Name, Position, Seniority, Fly_Hours)`
  - `flight(Flight#, Source_City, Dest_City, Date, No_Pass, Dep_Time, Arr_Time)`
  - `assign(ID#, Flight#)`

- Find the average number of passengers between two cities.

  ```sql
  SELECT Source_City, Dest_City, AVG(No_Pass)
  FROM flight
  GROUP BY Source_City, Dest_City;
  ```

- Find the number of flights assigned to each crew member (identified by name).

  ```sql
  SELECT Name, COUNT(Flight#)
  FROM assigns a, crew c
  WHERE a.ID# = c.ID#
  GROUP BY a.ID#, Name
  ```
SQL Aggregate Functions – Having Clause

- Aggregate operations using *Group By* and *Having*, which specifies conditions on each group created by *Group By*

- *Group By–Having* clause chooses groups based on some aggregate property of each group itself

- Example. Given the following relations:

  \[
  \text{crew(ID#}, \ \text{Name, Position, Seniority, Fly\_Hours)}
  \]

  \[
  \text{assign(ID#, Flight\#)}
  \]

  - Find the *average* number of *flying hours* of the crew members on any *flight* with more than one crew member

  \[
  \text{SELECT Flight\#, AVG(Fly\_Hours)}
  \]

  \[
  \text{FROM crew c, assigns a}
  \]

  \[
  \text{WHERE c.ID\# = a.ID\#}
  \]

  \[
  \text{GROUP BY Flight\#}
  \]

  \[
  \text{HAVING COUNT(*) > 1;}
  \]
SQL Aggregate Functions with Group-By and Having

- Given the following relation schemas:
  Account(Branch-Name, Account-Number, Balance)
  Customer(Customer-Name, Street, City)
  Depositor(Customer-Name, Account-Number)

- Find the average account balance at each branch
  
  SELECT Branch-Name, AVG(Balance)
  FROM account
  GROUP BY Branch-Name;

- Find the average account balance at each branch of which the average account balance is more than $1,200.

  SELECT Branch-Name, AVG(Balance)
  FROM account
  GROUP BY Branch-Name
  HAVING AVG(Balance) > 1200;
SQL Aggregate Functions with Group-By and Having

- Given the following relation schemas:
  
  Account(Branch-Name, Account-number, Balance)
  Customer(Customer-Name, Street, City)
  Depositor(Customer-Name, Account-Number)

- Find the number of tuples, i.e., customer, in the customer table.

  SELECT COUNT(*)
  FROM customer;

  Note: SQL does not allow the use of DISTINCT with COUNT(*)

- Find the average balance for each customer who lives in Harrison
  and has at least three accounts

  SELECT d.Customer-Name, AVG(Balance)
  FROM depositor d, account a, customer c
  WHERE d.Account-Number = a.Account-Number AND
       d.Customer-Name = c.Customer-Name AND
       City = ‘Harrison’
  GROUP BY d.Customer-Name
  HAVING COUNT(DISTINCT d.Account-Number) ≥ 3;