SQL

- DML (Data Manipulation Language)
  - Query language
  - Update language
    - Embedded SQL(SQL/J) and ODBC(JDBV)
      - necessary for application development
- DDL (Data Definition Language)
  - defines schema for relations
  - creats(modifies/destroys) database objects
- DCL (Data Control Language)
  - access control

SQL Data Types

- Values of attributes in SQL:
  - `integer` integer (32 bit)
  - `smallint` integer(16 bit)
  - `decimal(m,n)` fixed decimal
  - `float` IEEE float (32 bit)
  - `char(n)` character string (length n)
  - `varchar(n)` variable length string(at most n)
  - `date` year/month/day
  - `time` hh:mm:ss.ss
- **SQL is NOT case sensitive**

```
SELECT DISTINCT <results>
FROM            <tables>
WHERE           <condition>
```

$\{< \textbf{results} > | \exists < \textbf{unused} >. (\wedge < \textbf{tables} >) \wedge < \textbf{condition} >\}$

Variables vs. Attributes

- Relational Calculus uses *positional* notation, i.e.
  - `EMP(x,y,z)` is true whenever the x, y, and z components of an answer can be found as a tuple in the instance of EMP\
  - no need for attribute names
  - inconvenient for relations with high arity
- SQL uses ***corelations*** (tuple variables) and attribute names to assign default variable names to components of tuples:

- $R[\texttt{AS}]p$ in SQL stands for $R(p.a_1, \ldots, p.a_n)$ in RC where $a_1, \ldots, a_k$ are the attribute names declared for R.

Naming Attributes in the Results

- Results of queries $\Longleftrightarrow$ Tables
- What are the names of attributes in the result of a `SELECT` clause?
  - A single attribute: inherits the name
  - An expression: implementation dependent
- We can — and should — **explicitly** name the resulting attributes:
  - $\Longrightarrow$ `"<expr> AS <id>"` where `<id>` is the new name

Boolean Connectives

- Atomic conditions can be combined using **boolean connectives:**
  - `AND` (conjunction)
  - `OR` (disjunction)
  - `NOT` (negation)

Set Operations at Glance

$\Longrightarrow$ we can apply **set operations** on them:

- set union : $Q_1 \texttt{ UNION } Q_2$
  - the set of tuples in Q_1 or in Q_2
  - used to express "or"
- set difference : $Q_1 \texttt{ EXCEPT } Q_2$
  - the set of tuples in Q_1 but not in Q_2
  - used to express "and not"
- set intersection: $Q_1 \texttt{ INTERSECT } Q_2$
  - the set of typles in both Q_1 and Q_2
  - used to express "and" (redundant, rarely used).

Q_1 and Q_2 must have **union-compatible** signatures:

$\Longrightarrow$ same number and types of attributes

Nesting of Queries

- We can use SELECT Blocks (and other Set operations) as arguments of Set operations.
  - What if we need to use a Set Operation inside of a SELECT Block?
    - we can use distributive laws

      $\Longrightarrow (A \vee B) \wedge C \equiv (A \wedge C) \vee (B \wedge C)$

      $\Longrightarrow$ ofter **very** cumbersome(笨重)

- nest set operation inside a select block

  $\Longrightarrow$ common table expressions

Naming(Sub-)queries

- Syntax:

```
WITH fool [<opt-schema-1>]
     AS (<query-1-goes-here>),

     foon [<opt-schema-n>]
     As  (<quesy-n-goes-here>)

<query-that-uses-foot-...-foon-as-table-names>
```

The From clause revisited

- SQL-92 allows us to **inline** queries in the `FROM` clause:

  `FROM ... , ( <query-here> ) <id>, ...`

  $\Longrightarrow$ `<id>` stands for the result of `<query-here>`

  $\Longrightarrow$ unlike for base relations, `<id>` is **mandatory**.

`OR` and `UNION`

- A common mistake:
  - use of `OR` in the `WHERE` clause instead of the `UNION` operator
- e.g.

```
SELECT title
FROM publication, book, journal
WHERE publication.pubid = book.pubid
   OR publication.pubid = journal.pubid
```

- often works; but imagine there are no `book`s...(???)

`WHERE` subqueries

- Additional(complex) search conditions

  $\Longrightarrow$ query-based search predicates
- Advantages
  - simplifies writing queries with negation
- Drawbacks
  - complicated semantics (especially when duplicates are involved)
  - **very** easy to make mistakes

- **VERY COMMONLY** used to formulate queries

Overview of `WHERE` Subqueries

- presence/absence of a single value in a query

  `Attr IN (Q)`

  `Attr NOT IN (Q)`
- relationship of a value to some/all values in a query

  `Attr op SOME (Q)`

  `Attr op ALL (Q)`
- emptiness/non-emptiness of a query

  `EXISTS (Q)`

  `NOT EXISTS (Q)`

Example

- List all authors who always publish with someone else:

```
select distinct a1.name
from author a1, author a2
where not exists(
   select *
   from publication p, wrote w1
   where p.pubid = w1.publication
     and a1.aid = w1.author
     and a2.aid not in (
            select author
            from wrote
            where publication = p.pubid
              and author <> a1.aid
     )
);
```

How do we Modify a Database?

- Naive approch:

  ```
  DBSTART;
  r1: = Q1(DB);
  ...
  rk: = Qk(DB);
  DBCOMMIT;
  ```
- Not and acceptable solution in practice

Incremental Updtaes

- Tables are large but **updates are small** $\Longrightarrow$ Incremental updates
  - insertion of a tuples(`INSERT`)

    $\Longrightarrow$ constant tuple

    $\Longrightarrow$ results of queries
  - deletion of tuples (`DELETE`)

    $\Longrightarrow$ based on match of a condition
  - modification of tuples (`UPDATE`)

    $\Longrightarrow$ allows updating "in place"

    $\Longrightarrow$ based on match of a condition

SQL Insert

- Syntax:

```
INSERT INTO r[(a1,...,ak)]
       VALUES(v1,...vk)

INSERT INTO r (Q)
```

SQL Delete

- Syntax

```
DELETE FROM r
WHERE cond
```

SQL Update

- Syntax:

```
UPDATE r
SET <update statement>
WHERE <condition>
```

- Example

```
update author
set url = 'brics,dk/~david'
where aid in (
        select aid
        from author
        where name like 'Toman%'
);
```

Support for Transactions

- The DBMS guarantees noninterference (serializability) of all data access requests to tables in a database instance
- transaction starts with first **access** of the database

  $\Longrightarrow$ until it sees:
- `COMMIT:` make changes permanent

  `SQL> commit;`

  `Commit complete.`
- `ROLLBACK:` discard changes

  `SQL> rollback;`

  `Rollback complete`

Aggregation

- Standard and very useful extension of First-Order Queries
  - Aggregate(column) functions are introduced to
    - find number of tuples in a relation
    - add values of an attribute (over the whole relation)
    - find minimal/maximal values of an attribute
  - Can apply to groups of tuples that with equal values for (selected) attributes
  - Can **NOT** be expressed in Relational Calculus
- Syntax:

```
SELECT x1,...xk, agg1,...aggl
FROM Q
GROUP BY x1,...xk
```

- Restrications:
  - all attributes in the `SELECT` clause that are **NOT** in the scope of an aggregate function **MUST** appear in the `GROUP BY` clause.
  - `aggi` are of the form `count(y), sum(y), min(y), max(y), or avg(y)` whre y is an attribute of `Q` (usually not in the `GROUP BY` clause)

Operational Reading

1. pattition the input relation to groups with equal values of **grouping** attributes
2. on each of these partitions apply the **aggregate function**
3. collect the results and form the answer

- Example(count):
  - For each publication count the number of authors:

```
select publication, count(author)
from wrote
group by publication;
```

- Example(sum)
  - For each author count the number of article pages:

```
select author, sum(endpage-startpage+1) as pgs
from wrote, article
where publication = pubid
group by author;
```

  - not quite correct: it doesn't list 0 pages for author 3

`HAVING` clause

- The `HAVING` clause is mere *SYNTACTIC SUGAR* ... and can be replaced by a nested query and a `WHERE` clause
- Example:
  - List publications with exactly one author:

```
select publication, count(author)
from wrote
group by publication
having count(author) = 1;
```

- Example:
  - For every author count the number of books and articles:

```
select distinct aid, name, count(publication)
from author, (
  ( select distinct author, publication
    from wrote, book
    where publication = pubid )
  union all
  ( select distinct author, publication
    from wrote, article
    where publication = pubid) ) ba
where aid = author
group by name, aid;
```

Summary

- SQL covered so far:
  1. Simple SELECT BLOCK
  2. Set operations (`UNION, EXCEPT, INTERSECT`)
  3. Formulation of complex queries, nesting of queries, and views

4. Updating Data
5. Aggregation

- this covers pretty much all of the userful SQL DML

  $\Longrightarrow$ the Bad and Ugly coming next...