1 Syntax

SQL Queries have the generic form as follows:

```
SELECT [DISTINCT] List_of_attributes
FROM List_of_tables
[ WHERE Condition]
[ ORDER BY List_Of_Ordering_Attributes ]
[ GROUP BY List_Of_Grouping_Attributes
[ HAVING Having_Condition] ]
[ UNION [All] | INTERSECT [All] | EXCEPT [All]
SUB_QUERY ]
```

where the part appears between [ and ] are optional. A (simple) description of components are given below:

1. **List_of_Attributes**: a list of

   (a) attribute name (e.g. maker)
   (b) attribute name AS new_name (this is used for renaming)
   (c) aggregation expression (e.g. SUM(price))
   (d) arithmetic expression (e.g. price*2)

   which are separated by comma (* stands for ALL). *This list specifies what attributes does the output relation have.*

2. **List_of_Tables**: a (separated by comma) list of relation names (or table names) (e.g. product, PC); this can also contain a subquery; *This list specifies the sources of the information used for the query.*

3. **Condition**: a Boolean expression built using
(a) attribute names,
(b) logical operators: AND, OR, NOT
(c) arithmetic operators (+,-,*,) on numeric valued attributes
(d) comparison operators (¿,¡,=,¿=,¡=) or special comparison operator (e.g. LIKE) for strings

*This condition specifies the tuples from the relations that the query specifier is interested in.*

In constructing the condition, special care needs to be made when working with NULL values which indicate unknown value (the price is unknown, the birthdate is unknown, etc.):

- Arithmetic operations with NULL as one of the arguments yield NULL result, e.g., 50 + NULL gives NULL, 10 * NULL gives NULL.
- Boolean operations with NULL might yield UNKNOWN truth: see table on three-valued logic in the book (table 6.2); for instance, A > NULL gives UNKNOWN.

4. List Of Ordering Attributes: *list of attributes which are used for ordering the output*; this is similar to the list $L$ in the sorting operator $\tau_L(R)$ of extended operator of relational algebra.

5. List Of Grouping Attributes: *list of attributes which are used for grouping the data for aggregation operations*; this is similar to the list of grouping attributes occurring in $L$ of the operator $\gamma_L(R)$.

6. Having Condition: construction is similar to the condition in the where clause; the effect of this part is to select the groups that satisfy this condition.

7. SUB_QUERY: another query.

8. DISTINCT: used to specify the set operation instead of the bag operation that is the default in SQL.

## 2 SQL queries and Relational Algebra

Each relational algebra expression (in extended operators) can be translated in a SQL query that yields the same result (the output contains the same set of tuples). This will be demonstrated through examples in this section. We will use the running example of a database with the following relations:

```sql
product (maker, model, type)
pc (model, speed, ram, hd, cd, price)
laptop (model, speed, ram, hd, screen, price)
printer (model, color, type, price)
```
2.1 Projection/Selection

The expression $\pi_{A_1,...,A_n}(R)$ and the SQL query:

```
SELECT A_1,...,A_n FROM R
```

will give the same answer. For example, $\pi_{\text{maker}}(\text{product})$ and `SELECT maker FROM product` will give the list of all producers in the database.

The query `SELECT DISTINCT maker FROM product` will list the list of all producers without duplication.

The expression $\sigma_{\text{Condition}}(R)$ and the SQL query:

```
SELECT * FROM R WHERE Condition
```

will give the same answer. For example, $\sigma_{\text{maker}='A'}(\text{product})$ and `SELECT * FROM product where maker='A'` will give the list of all the products (model, type) produced by producer ‘A’.

We can combine selection and projection as in relational algebra. Examples:

```
select model from pc where speed>=200 and price < 2500
```

is equivalent to $\pi_{\text{model}}(\sigma_{\text{speed}>=200 \text{ and price}<2500}(pc))$

```
select model from pc where cd like ’8%’
```

is equivalent to $\pi_{\text{model}}(\sigma_{\text{cd}=’8’}(pc))$ where the % stands for any string;

The following query uses the renaming and ordering features:

```
select model, speed as megahertz, ram, hd as gigabytes
from pc
where price<2500
order by speed, ram
```

2.2 Join, Union, Difference, and Intersection

The natural join between two relations $R$ and $S$, $R \bowtie S$ can be specified in the WHERE clause (the condition) by having a condition of the form $R.A_1 = S.A_1 \text{ AND } R.A_2 = S.A_2 \text{ AND } \ldots$. For example, to join the relations product and laptop and get the maker who produces laptop with the hard disk larger than 0.3 GB, we write

```
select maker, speed
from product, laptop
where product.model=laptop.model and hd>='0.3'
```
which is equivalent to
\[ \pi_{\text{maker}, \text{speed}}(s_{hd} \geq 0.3 (\text{product} \bowtie \text{laptop})). \]

To list the price and model of PC produced by maker ‘B’ we write:

```sql
select product.model, price  
from pc, product where maker='B' and product.model=pc.model
```

which is the same as
\[ \pi_{\text{model}, \text{price}}(s_{\text{maker}='B'} (\text{product} \bowtie \text{pc})). \]

**NOTE:** In SQL queries, we use the notation `tablename.attributename` to disambiguate attributes when relations involving in the query have the same attribute name. For instance, `product.model` specifies that the value should be taken from the attribute `model` of the relation `product`.

To get the list of products and their price produced by maker ‘B’ we use UNION:

```sql
select product.model, price  
from pc, product where maker='B' and product.model=pc.model  
union  
select product.model, price  
from laptop, product where maker='B' and product.model = laptop.model  
union  
select product.model, price  
from printer, product where maker='B' and printer.model=product.model
```

This is equivalent to \[ \pi_{\text{model}, \text{price}}(s_{\text{maker}='B'} (\text{product} \bowtie \text{pc})) \cup \pi_{\text{model}, \text{price}}(s_{\text{maker}='B'} (\text{product} \bowtie \text{laptop})) \cup \pi_{\text{model}, \text{price}}(s_{\text{maker}='B'} (\text{product} \bowtie \text{printer})). \]

To get the list of producer who produces both PC and laptop we use INTERSECTION (INTERSECT in SQL):

```sql
select maker from pc, product where product.model=pc.model  
intersect  
select maker from laptop, product where product.model=laptop.model
```

This is equivalent to \[ \pi_{\text{maker}}(\text{product} \bowtie \text{pc}) \cup \pi_{\text{maker}}(\text{product} \bowtie \text{laptop}). \]

To get the list of producer who produces PC but do not produce laptop we use SET DIFFERENCE (EXCEPT in SQL):

```sql
select maker from pc, product where product.model=pc.model  
except  
select maker from laptop, product where product.model=laptop.model
```

This is equivalent to \[ \pi_{\text{maker}}(\text{product} \bowtie \text{pc}) \setminus \pi_{\text{maker}}(\text{product} \bowtie \text{laptop}). \]

If we want to have a bag (keep the duplications) in the above operations, we should write **ALL** after `union`, **INTERSECT**, or `except`, respectively.
2.3 Aggregation and Grouping

Aggregation operators: SUM, AVG, MIN, MAX, COUNT. Relational algebra expression involving these operators $\gamma_L(R)$ can be translated into a SQL query whose List_of_Attributes as $L$ as follows:

- put all the grouping attributes occurring in $L$ into the list of grouping attributes of the corresponding SQL query,

for example, $\gamma_{hd.AVG(speed)}(pc)$ is translated into

```sql
select hd, AVG(speed)
from pc
  group by hd
```

Find the maker(s) of PC(s) with the fastest processor among all those PC’s that have the smallest amount of RAM

```sql
select maker, pc.model, price, speed, ram
  from pc, product where pc.model = product.model
```

```sql
select maker, pc.model, price, speed, ram
  from pc, product where pc.model = product.model order by ram, speed
```

```sql
select maker, pc.model, price, speed, ram
  from pc, product where pc.model = product.model
  and pc.speed = (select max(speed) from pc
      group by ram having ram = (select min(ram) from pc))
  and pc.ram = (select min(ram) from pc)
```

3 Creating a Relation Schema in SQL

3.1 Data types

The basic data type:

1. Character string: CHAR(n) (fixed length $n$) or VARCHAR(n) (up to $n$ characters) where $n$ is an integer.
2. Bit string: BIT(n) (fixed length $n$) or VARBIT(n) (up to $n$ bits) where $n$ is an integer.
3. BOOLEAN: TRUE, FALSE, or UNKNOWN.
4. INT or INTEGER: integer.
5. FLOAR or REAL or DECIMAL(n,d): real values.
6. DATE and TIME: date and time.
3.2 Simple Table Declaration

Syntax:

```
CREATE TABLE relation_name(
  attribute_name_1 type_of_attribute_1,
  attribute_name_2 type_of_attribute_2,
  ...
  attribute_name_m type_of_attribute_m
);
```

where the `attribute_name_i` and `type_of_attribute_i` is the name and the type of the \( i^{th} \) attribute, respectively. For example, we use

```
CREATE TABLE produce(
  maker varchar(1),
  model integer,
  type varchar(10)
);
```

to create the table `product`.

We can specify a default for an attribute when creating the table by adding 'DEFAULT attribute_value' after the type specification of the attribute. For example,

```
CREATE TABLE produce(
  maker varchar(1) default 'U',
  model integer,
  type varchar(10)
);
```

specifies that the maker should have the value 'U' if it was not specified by others.

3.3 Database Modifications

This includes: insertion, deletion, and update the tuples of a relation. The SQL commands for these activities are: INSERT, DELETE, and UPDATE.

**Insertion.** Syntax:

```
INSERT INTO Relation_Name (att1,...,attn) VALUES (v1,...vn)
```

where

1. Relation_Name: the name of the relation that will be inserted a new tuple
2. (att1,...,attn): the (might be incomplete) list of attributes of Relation_Name
3. (v1,...vn): the list of values assigned to the attributes in the attribute list (att1,...,attn); the value \( v_i \) is assigned to the attribute \( a_{ti} \).
Example: to insert the tuple \( ('A',1000,'pc') \) into the table product, we write

```sql
INSERT INTO product (maker,model,type) VALUES ('A',1000,'pc')
```

this can be shorten to

```sql
INSERT INTO product VALUES ('A',1000,'pc')
```

If we write

```sql
INSERT INTO product (model,type) VALUES (1000,'pc')
```

then a new tuple is added in which maker has the NULL value or some default value that is specified when we create the table;

We can get a set of tuples from a relation and insert them into another relation by using the following form

```sql
INSERT INTO Relation_Name (att_list) QUERY
```

where \( \text{QUERY} \) is a query described earlier, \( \text{att_list} \) is the list of attributes from \( \text{Relation}_\text{Name} \); \( \text{QUERY} \) must return a relation (a set/bag of tuples) whose attributes have the same type as those occurring in \( \text{att_list} \). For example, we can create a relation maker with one attribute producer_name by

```sql
CREATE TABLE maker (producer_name varchar(1))
```

and insert into this table all the makers from the relation product by the command

```sql
INSERT INTO maker (producer_name)
SELECT DISTINCT maker FROM product
```

**Deletion. Syntax:**

```sql
DELETE FROM Relation_Name WHERE Condition
```

where

1. Relation_Name: the name of the relation from which tuples will be deleted;
2. Condition: specifies which tuples will be removed.

Example: To delete all the products of maker 'B' in the table product, we write

```sql
DELETE FROM product WHERE maker = 'B'
```

**Update. Syntax:**

```sql
UPDATE Relation_Name SET new_value_assignment WHERE Condition
```
where

1. Relation_Name: the name of the relation from which information will be updated;

2. new_value_assignment: a list of formulas, each sets an attribute of the relation equal to the value of an expression or constant;

3. Condition: specifies which tuples will be updated.

Example: To change all products of maker 'B' into products of 'A', we write

```
UPDATE product SET maker = 'A' WHERE make='B'
```

## 4 Views

View – a different presentation of a relation; does not exist physically (will not exist after the session is closed); defined by a query (expression);

### 4.1 Declaring/Querying Views

```
CREATE VIEW View_Name AS View_Definition
```

where

1. View_Name: the name that will be used to refer to the view created by the command;

2. View_Definition: the expression that specifies the tuple belonging to the view

For example, we want to have the list of pc with ram > 32 in a view, we write

```
CREATE VIEW pcRamMoreThan32 AS
    SELECT * FROM PC WHERE ram > 32
```

After this command, we can use `pcRamMoreThan32` like product, pc, etc. However, if we quit and reenter db2, the relation will no longer exist. For example, we can use `pcRamMoreThan32` to find the fastest pc that has more than 32 Mb ram:

```
SELECT model, speed, ram
    FROM pcRamMoreThan32
    WHERE speed = max(speed)
```
4.2 Modifying View

Only *updatable views* can be modified. When it is done, this is translated into an equivalent modification on a base table (the table stored physically). The important characters of updatable views are:

1. The view must be defined by a SELECT clause (not SELECT DISTINCT) that selects tuples from a relation \( R \);
2. The WHERE clause must not involve the relation \( R \);
3. The list in the SELECT clause must include enough attributes that for every tuple inserted into the view, we can fill other attributes out with NULL values or proper default and have a tuple of the base relation that will yield the inserted tuple of the view;

Consider the view:

```sql
CREATE VIEW pcModelPriceRamMoreThan32 AS
  SELECT model, price FROM PC WHERE ram > 32
```

This will give us a relation with two attributes model and price. This satisfies the first two conditions of an updatable view but not the third one. To see why, we can look at the command

```
INSERT INTO pcModelPriceRamMoreThan32 VALUES (5000,4599)
```

This command attempts to insert into the view `pcModelPriceRamMoreThan32` a new tuple with model=5000 and price=4599. The problem is that `ram` is not an attribute of this view, and hence, the tuple – that should be inserted into PC – will have NULL value for ram, which does not meet the criteria that its ram is greater than 32. (NULL compares with 32!)

However, the view `pcRamMoreThan32` is an updatable view. We can do

```
INSERT INTO pcRamMoreThan32 (model, price) VALUES (5000, 4599)
```
and a new tuple will be inserted into `pcRamMoreThan32` and the table pc.

Deletions and updates can also be done on updateable views. They will be translated into a command that affects the base table as well.