SQL: Structured Query Language
SELECT Statement

SELECT Statement

The core of SQL, is the SELECT statement. Basic SELECT statement looks as follows:

SELECT [DISTINCT] *select-list*
FROM *from-list*
[WHERE *qualification*]

More complex SELECT statements will be studied later.

- *from-list* contains the list of database relations from which the data is to be retrieved.
- *select-list* contains the list of relation attributes (possibly modified) to be returned in the answer to the query.
- *qualification* contains the conditions which must be satisfied by a database record to be put into the answer set.

Evaluation of SELECT statement:

1. *from-list* defines a *cartesian product* and/or *joins* of all relations in it.
2. *qualification* defines *selection* conditions on the data.
3. *select-list* defines the final look of the output, i.e., the *projection* attributes.
4. DISTINCT specifies duplicate elimination in the final answer set (default for SELECT QUERY: no duplicate elimination).

SQL SELECT Statement and Relational Algebra

As one can guess from its evaluation, SELECT statement implements Relational Algebra operations *selection*, *projection*, *cartesian product* and *join*.

SELECT and Selection

Relational algebra operation: $\sigma_C(R)$
SELECT statement

```
SELECT DISTINCT *
FROM R
WHERE C
```

Note: "*" is a special notation for the selection list that includes *all available attributes*.
Example: RA: $\sigma_{salary>20000}(Employee)$
SQL:

```
SELECT DISTINCT *
FROM Employee
WHERE salary > 20000
```

SELECT and Projection

Relational algebra operation: $\pi_F(R)$
SELECT statement

```
SELECT DISTINCT F
FROM R
```

Note: Relational algebra operations return sets (with no duplicates), hence to represent *true* relational algebra projection, we need to use DISTINCT here. While we may ignore this sometimes, please, keep this fact in mind.
Example: RA: $\pi_{name,salary,position}(Employee)$
SQL:

```
SELECT DISTINCT
    name,
    salary,
    position
FROM Employee
```

SELECT and Cartesian Product

Relational algebra operation: $R1 \times R2$
SELECT statement

```
SELECT *
FROM
    R1,
    R2
```

Note: As mentioned above, more than one relation in the from list represents Cartesian Product.
Example: RA: $Employee \times Client$
SQL:

```
SELECT *
FROM
    Employee,
    Client
```

SELECT and Cartesian Product (Style Join)

Relational algebra operation: $R1 \bowtie_{R1.A=R2.B} R2$
SELECT statement

```
SELECT *
FROM
    R1,
    R2
WHERE R1.A = R2.B
```

Note: For clarity, a simple equijoin is used here. This conversion can be extended to other types of join.
Example: RA: $Employee \bowtie_{Employee.manages=Client.id} Client$
SQL:

```
SELECT *
FROM
    Employee,
    Client
WHERE Employee.manages = Client.id
```

SELECT and Join

Relational algebra operation: $R1 \bowtie_{R1.A=R2.B} R2$
SELECT statement

```
SELECT *
FROM
    Relation1 R1
    JOIN Relation2 R2 ON R1.A = R2.B
```

Note: Most SQL implementations requires that tables used in a JOIN get a *table alias*. In this example, the tables *Relation1* and *Relation2* were alised to *R1* and *R2* respectively.
Example: RA: $Employee \bowtie_{Employee.manages=Client.id} Client$
SQL:

```
SELECT *
FROM
    Employee E
    JOIN Client C C.id = E.manages
```

From-list

from-list of the SELECT statement has the following format:

TableName [*TableAlias*] [, . . .]

- *TableName* is a name of the existing database relation.
- *TableAlias* also known as *range variable* is an identifier that can be used *instead* of *TableAlias*.

While there can be two *TableNames* that have the same value in a from-list, all *TableAliases* are unique! Table aliases allow representation of self-joins in SQL.

Example

Find all employees in John Smith's department who have a bigger salary then John and output their records side by side.

```
SELECT *
FROM
    Employee E1,
    Employee E2
WHERE
    E1.department = E2.department
    AND E1.salary < E2.salary
    AND E1.name = 'John Smith'
```

Select-list

A simple select-list is a comma-separated sequence of attribute names. However, more complex expressions can appear in a select list.

- *: as mentioned above, this is a shortcut for "all attributes of all relations involved in the query in their natural order". If a '*' is used, no other attributes may be in the select-list.
- *AttributeName*: a simple attribute name can be part of a select-list if the attribute name alone is sufficient to uniquely identify the attribute among all attributes involved in the query.
- *TableName.AttributeName*: if two different relations involved in the query have attributes with the same name, table name must preface the attribute name to guarantee unambiguous identification.
- *TableAlias.AttributeName*: whenever table alias is defined for a relation *it is always safe* to reference the attribute in this manner. This is also the only way to indicate the exact attribute to be included into select-list in self-joins.
- *Expression*: An expression over different attributes involved in a query is can also be returned in select-list.
- *AttributeSpec* AS *NewAttributeName*: this form is used to rename the attribute in the result of the query. It can be used to give names to the attributes that are results of expressions as well as to replace *TableName.AttributeName* names with simpler attribute names.

- *Aggregate Expressions*: SELECT queries also allow for return of aggregate expressions (sums, averages, minimums and maximums, etc.) of attributes. More on that later.

Set Operations in SQL

SELECT statement incorporates a combination of *selection*, *projection*, and *cartesian product* operations (which also enables *join*). It does not, by itself, allow us to express set operations in relational algebra. The latter is done using three special SQL statements.

Union in SQL

The format of the union statement is:

Expression `UNION` *Expression*

Here, *Expression* is any expression that resolves as a relational table. Typically, it will be a SELECT statement.

The result of the statement is a relational table which is a union of the two table represented by the right-hand side and left-hand side expressions. Duplicates are automatically eliminated.

For example, if `A` and `B` are two relations with the same schema, to compute their union, we use the following query:

```
SELECT * FROM A
UNION
SELECT * FROM B
```

Note: Some SQL implementations (such as MySQL) have a syntax to support a UNION-like operations that supports duplicates. (*UNION ALL* in MySQL.)

Difference in SQL

MySQL does not support a single difference operator.[1]

To replicate set difference, a semi-join can be used. (We will discuss SQL joins in detail later). For example, to compute the difference between two tables `A` and `B` with the same schema, write the following query:

```
SELECT *
FROM A LEFT JOIN B USING(someColumn)
WHERE B.someColumn IS NULL
```

Intersection in SQL

Like difference, MySQL does not support a single intersection operator.[2]

To replicate set intersection, a simple natural join can be used (note that if you are finding the intersection, the schemas should already match). For example, to compute the intersection between two tables `A` and `B` with the same schema, write the following query:

```
SELECT *
FROM A NATURAL JOIN B
```

[1] Some other SQL servers such as Oracle SQL and PostgreSQL do support a difference operator usually called EXCEPT or MINUS.

[2] Some other SQL servers such as Oracle SQL and PostgreSQL do support a difference operator usually called INTERSECT.

Examples

1. Different attribute identifiers. *Find all books that were borrowed at least twice in 2002 and output the names of the two borrowers in each record in chronological order.*

```
SELECT
    title,
    Book.id,
    P1.name,
    P2.name
FROM
    Books,
    Loans L1,
    Loans L2,
    Patrons P1,
    Patrons P2
WHERE
    Books.id = L1.bId
    AND Books.id = L2.bId
    AND P1.id = L1.pId
    AND P2.id = L2.pId
    AND L1.date < L2.date
    AND L1.date > DATE('2001-12-31')
```

   Note: This query is a 5-way join that includes two self-joins: on Patrons and Loans. *title* attribute is unambiguous as it exists only in Books. Book qualifier is needed for *Book.id* because Patrons relation also has *id* field. Finally, $P1$ and $P2$ qualifiers for the two *name* attributes are needed to indicate which attribute comes from which copy of the Patrons relation.

2. Expressions as columns. *Output for each employee the sum of his/her salary and bonus.*

```
SELECT
    name,
    salary + bonus
FROM Employee
```

3. Renaming output columns *Output for each employee the sum of his/her salary and bonus.*

```
SELECT
    name AS Employee_Name,
    salary + bonus as Compensation
FROM Employee
```

   Note: same query as above, only the first field is renamed "Employee_Name" and the second field is given a new name "Compensation".