Introduction to SQL

SQL Data Definition

1. Basic Types
   - `char(n)`: A fixed-length character string with user-specified length n. The full form, character, can be used instead. *Store the fixed length of the string. If the actual length of the string is less than n, spaces will be appended to the string to make it n characters long.*
   - `varchar(n)`: A variable-length character string with user-specified maximum length n. The full form, character varying, is equivalent. *No space will be added if the length of attribute is less than n.*
   - `int`: An integer (a finite subset of the integers that is machine dependent). The full form, integer, is equivalent.
   - `smallint`: A small integer (a machine-dependent subset of the integer type).
   - `numeric( p, d)`: A fixed-point number with user-specified precision. The number consists of p digits (plus a sign), and d of the p digits are to the right of the decimal point. Thus, numeric(3,1) allows 44.5 to be stored exactly, but neither 444.5 or 0.32 can be stored exactly in a field of this type.
   - `real, double precision`: Floating-point and double-precision floating-point numbers with machine-dependent precision.
   - `float(n)`: A floating-point number, with precision of at least n digits.
2. Basic Schema Definition
   1. Define an SQL relation - `create table`

      > CREATE TABLE *table_name* (
      > *column1 datatype*,
      > *column2 datatype*,
      > *column3 datatype*,
      > ....
      > <integrity-constraint 1 >,
      > ...,
      > <integrity-constraint k >
      >);

      Integrity Constraints:

      1. **primary key** (A_{j1} , A_{j2} , . . . , A_{jm}): The primary-key specification says that attributes A_{j1} , A_{j2} , . . . , A_{jm} form the primary key for the relation.
         - The primary key attributes are required to be *nonnull* and *unique*
      2. **foreign key** (A_{k1} , A_{k2} , . . . , A_{kn}) **references** s: The foreign key specification says that the values of attributes (A_{k1} , A_{k2} , . . . , A_{kn}) for any tuple in the relation must correspond to values of the primary key attributes of some tuple in relation s.
      3. **not null**: The not null constraint on an attribute specifies that the null value is not allowed for that attribute;
   2. Load data into relation - `insert`

> ***INSERT INTO*** *table_name* (*column1*, *column2*, *column3*, ...)
> ***VALUES*** (*value1*, *value2*, *value3*, ...);
>
> ***INSERT INTO*** *table_name*
> ***VALUES*** (*value1*, *value2*, *value3*, ...);

3. Deletes tuples from a relation - `delete`

> ***DELETE FROM*** *table_name* ***WHERE*** *condition*;

4. Remove a relation from an SQL database - `drop table`

> ***DROP TABLE*** *table_name*;

5. Add attributes to an existing relation - `alter table`
   - Can be used to add, delete, or modify columns in an existing table.
   - Can also be used to add and drop various constraints on an existing table.

> ALTER TABLE *table_name*
> ADD *column_name datatype*;
>
> ALTER TABLE *table_name*
> DROP COLUMN *column_name*;

3. Basic Structure of SQL Queries

`select`, `from`, `where`

1. Queries on a Single Relation

```
select name from instructor;
```

   - Duplicate retention is not the default. The return result can contain duplicate. If we want to remove the duplicate, we have to specify the attribute to be distinct by the keyword **distinct**

```
select distinct dept_name from instructor;
```

   - The **select** clause may contain arithmetic expressions involving the operators `+`, `-`, `*`, and `/` operating on constants or attributes of tuple.

```
select ID , name, dept name, salary * 1.1
from instructor;
```

   - The **where** clause allows us to select only those rows in the result relation of the **from** clause that satisfy a specified predicate.

```
--Find the names of all instructors in the Computer Science
department who have salary greater than $70,000.
select name
from instructor
where dept name = 'Comp. Sci.' and salary > 70000;
```

2. Queries on Multiple Relations

The role of each clause is as follows:

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The where clause is a predicate involving attributes of the relation in the from clause.

A typical SQL query has the form

> select A_1 , A_2 , . . . , A_n
>
> from r_1 , r_2 , . . . , r_m
> where P;

Each A_i represents an attribute, and each r_i a relation. P is a predicate. If the **where** clause is omitted, the predicate P is **true**.

- Since the same attribute name may appear in both r_i and r_j , we prefix the name of the relation from which the attribute originally came, before the attribute name, e.g., `Movie.id`, `Actor.id`

3. The Natural Join

The **natural join** operation operates on two relations and produces a relation as the result.

```
-- The followings are equivalent
select name, course id
from instructor, teaches
where instructor.ID = teaches.ID ;
-----------------------------------------
select name, course id
from instructor natural join teaches;
```

> **select** A_1 , A_2 , . . . , A_n
> **from** r_1 **natural join** r_2 **natural join** . . . **natural join** r_m
> **where** P;

- To provide the benefit of natural join while avoiding the danger of equating attributes erroneously, SQL provides a form of the natural join construct that allows you to specify exactly which columns should be equated. - `join..using`

```
select name, title
from (instructor natural join teaches) join course using (course
id);
```

4. Additional Basic Operations

1. The Rename Operation - `as`

When we should derive name:

- First, two relations in the from clause may have attributes with the same name, in which case an attribute name is duplicated in the result.
- Second, if we used an arithmetic expression in the select clause, the resultant attribute does not have a name.
- Third, even if an attribute name can be derived from the base relations, we may want to change the attribute name in the result.

The as clause is particularly useful in renaming relations. Why we want to rename a relation:

- Replace a long relation name with a shortened version that is more convenient to use elsewhere in the query.
- We rename a relation such that we can compare tuples in the same relation

```
--Find the names of all instructors whose salary is greater than at
least one instructor in the Biology department.
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name = 'Biology';
```

2. String Operations
   - Equality operation on strings is case sensitive.
   - SQL permits a variety of functions on character strings.
   - Pattern mathching can be performed on strings, using the operator `like`.
     - Percent (%): The % character matches any substring.
       - 'Intro%' matches any string beginning with "Intro".
     - Underscore (_): The character matches any character.
       - '_ _ _' matches any string of exactly three characters.
       - '_ _ _%' matches any string of at least three characters.
3. Attribute Specification in Select Clause - `*`

   The asterisk symbol `*` can be used in the select clause to denote "all attributes."
4. Ordering the Display of Tuples - `order by`

   The `order by` clause causes the tuples in the result of a query to appear in sorted order.

```
select name
from instructor
where dept name = 'Physics'
order by name;
```

   - By default, the **order by** clause lists items in ascending. To specify the sort order, we may specify **desc** for descending order or **asc** for ascending order.

```
-- if several instructors have the same salary, then we order it by name
in ascending order
select *
from instructor
order by salary desc, name asc;
```

5. Where Clause Predicates

   SQL includes a **between** comparison operator to simplify **where** clauses that specify that a value be less than or equal to some value and greater than or equal to some other value.

```sql
-- The followings are equivalent
select name
from instructor
where salary between 90000 and 100000;

select name
from instructor
where salary <= 100000 and salary >= 90000;
```

5. Set Operations
    1. The Union Operation
        - The **union** operation automatically *eliminates duplicates*, unlike the select clause.

        ```sql
        (select course id
         from section
         where semester = 'Fall' and year= 2009)
        union
        (select course id
         from section
         where semester = 'Spring' and year= 2010);
        ```

        - If we want to retain all duplicates, we must write **union all** in place of **union**

        ```sql
        (select course id
         from section
         where semester = 'Fall' and year= 2009)
        union all
        (select course id
         from section
         where semester = 'Spring' and year= 2010);
        ```

    2. The Intersect Operation
        - The **intersect** operation automatically eliminates duplicates.
        - If we want to retain all duplicates, we must write **intersect all** in place of **intersect**
    3. The Except Operation
        - The **except** operation outputs all tuples from its first input that do not occur in the second input; that is, it performs *set difference*.
        - It remove duplicates automatically, and if we want to retain all duplicates, we must write **except all** in place of **except**.
6. Null Values
    - SQL treats as unknown the result of any comparison involving a null value. This creates a third logical value in addition to true and false.
    - Since the predicate in a where clause can involve Boolean operations such as and, or, and not on the results of comparisons, the definitions of the Boolean operations are extended to deal with the value unknown.
        - **and**: The result of true and unknown is unknown, false and unknown is false, while unknown and unknown is unknown.
        - **or**: The result of true or unknown is true, false or unknown is unknown, while unknown or unknown is unknown.
        - **not**: The result of not unknown is unknown.

- If the where clause predicate evaluates to either false or unknown for a tuple, that tuple is not added to the result.

7. Aggregate Functions
    1. Basic Aggregation
        - Average: **avg**
        - Minimum: **min**
        - Maximum: **max**
        - Total: **sum**
        - Count: **count**
    2. Aggregation with Grouping - `group by`
        - The attribute or attributes given in the group by clause are used to form groups.
        - Tuples with the same value on all attributes in the group by clause are placed in one group.
    3. The Having Clause
        - The `having` clause is used to state a condition that applies to groups rather than to tuples.
        - This condition does not apply to a single tuple; rather, it applies to each group constructed by the group by clause.
        - The meaning of a query containing aggregation, group by, or having clauses is defined by the following sequence of operations:
            1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
            2. If a where clause is present, the predicate in the where clause is applied on the result relation of the from clause.
            3. Tuples satisfying the where predicate are then placed into groups by the group by clause if it is present. If the group by clause is absent, the entire set of tuples satisfying the where predicate is treated as being in one group. (**where** cannot put after **group by**?)
            4. The having clause, if it is present, is applied to each group; the groups that do not satisfy the having clause predicate are removed.
            5. The select clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.
8. Nested Subqueries

    A subquery is a **select-from-where** expression that is nested within another query.

    1. Set Membership
        - The **in** connective tests for set membership, where the set is a collection of values produced by a select clause.
        - The **not in** connective tests for the absence of set membership.
    2. Set Comparison
        - The phrase "greater than at least one" is represented in SQL by **> some**.
            - **= some** is identical to **in**, whereas **<> some** is *not the same as* **not in**.
        - The construct **> all** corresponds to the phrase "greater than all."
            - It can be used in finding the "largest", "maximum".
    3. Test for Empty Relations
        - The **exists** construct returns the value true if the argument subquery is *nonempty*.
        - The **not exists** construct returns the value true if the argument subquery is *empty*.

        - We can use the **not exists** construct to simulate the set containment (that is, superset) operation: We can write "relation A contains relation B" as "not exists (B except A)."
    4. Test for the Absence of Duplicate Tuples
        - The **unique** construct returns the value true if the argument subquery contains no duplicate tuples.
            - The **unique** can be used to find something "at most once".
            - The **not unique** can be used to find something "at least twice".
    5. Subqueries in the From Clause
        - Any **select-from-where** expression returns a relation as a result and, therefore, can be inserted into another **select-from-where** anywhere that a relation can appear.
    6. The with Clause
        - The **with** clause provides a way of defining a temporary relation whose definition is available only to the query in which the **with** clause occurs.

```
with max budget (value) as
    (select max(budget)
     from department)
 select budget
 from department, max budget
 where department.budget = max budget.value;
```

9. Modification of the Database
    1. Deletion

> **delete from** r **where** P;

        - The **delete** statement first finds all tuples t in r for which P(t) is true, and then deletes them from r.
        - The **where** clause can be omitted, in which case all tuples in r are deleted.
    2. Insertion

> **insert into** t **values** (attribute$_1$, attribute$_2$,..., attribute$_n$);

        - To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.
    3. Updates

> **update** t **set** attribute_name = value **where** P;

        - **update** changes a value in a tuple without changing all values in the tuple.
        - SQL provides a **case** construct that we can use to perform both the updates with a single update statement, avoiding the problem with the order of updates.

> **CASE**
> **WHEN** *condition1* **THEN** *result1*
> **WHEN** *condition2* **THEN** *result2*
> **WHEN** *conditionN* **THEN** *resultN*
> **ELSE** *result*
> **END**;