Chpater 3: Introduction to SQL

Contents

1 Overview of SQL

1) Developed as *Sequel* by IBM as part of their System R project in the early 1970s. Evolved into SQL (Structured Query Language).
2) Various standards were published for SQL such as SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011, and SQL:2016.
3) SQL has several parts:
   a) **Data-definition Language (DDL):** Allows specification of
      i) Defining relation schemas
      ii) Deleting relations
      iii) Modifying relation schemas
      iv) Integrity constraints
      v) View definitions
      vi) Authorization
   b) **Data Manipulation Language (DML):** Provides commands for
      i) Querying information from databases
      ii) Insert, delete and modify tuples in a database
   c) **Transaction control:** Specify begin and end points of transactions
   d) **Embedded/dynamic SQL:** Define how SQL can be embedded into general-purpose languages.
4) **Note:** Comments in SQL are written in between `/* ... */` or are preceded by '–'.

2 SQL Data Definition

2.1 Basic Types

1) **char(n):** A fixed length character string of user-specified length n. Full form **character**. Padded with whitespace at the end, if any.
2) **varchar(n):** A variable-length character string of maximum length n specified by the user. Full form **character varying**.
3) **int:** An integer. Full form **integer**. Machine-dependent size.
4) **smallint:** Small integer. Machine-dependent size.
5) **numeric(p, d):** A fixed-point number with a user-specified precision. Number has p digits and d of the p digits are to the right of the decimal point.
6) **real, double precision:** Floating-point and double-precision floating-point numbers. Machine-dependent precision.
7) **float(n):** A floating point number with at least n digit precision.

Each type includes a special **null** value, indicating an absent or unknown value. To store multilingual data using the Unicode representation, use the **nvarchar** type.

2.2 Basic Schema Definition

SQL relation is defined by **create table** command. Syntax:

```
create table tableName
(
    attr_1 domain_1,
    attr_2 domain_2,
    ...,
    attr_n domain_n,
    constr_1,
    constr_2,
    ...,
    constr_m
);
```

NOTE: ALL statements in SQL MUST end with a **semicolon** (;).

Integrity constrains supported by SQL:

1) **primary key**$(A_{j_1}, A_{j_2}, \ldots, A_{j_m})$**:** Specifies the attributes in brackets that form the primary-key for the relation. All attributes should be non-null and unique. Optional but recommended constraint.
2) **foreign key**$(A_{k_1}, A_{k_2}, \ldots, A_{k_n})$ **references** s**:** Values of attributes in brackets for any tuple

in the relation must correspond to primary key attributes of some tuple in relation *s*.
3) **not null:** Specifies that the null value is not allowed for that attribute. Exculdes the **null** value from the domain of the attribute.

Note: Tuples that violate integrity constraints are not inserted in the relation.

To remove a relation from a database

1) Use **drop table** *r*. This command deletes all information about table *r*.
2) In contrast, **delete from** *r* does not delete the table *r*, but deletes all tuples in *r*.
3) After a table is dropped, the **create table** command must be used to insert tuples into it.

To alter the schema of an existing relation, the **alter table** command is used.

1) To add an attribute: **alter table** *r* **add** *A D*, where:
   a) *r* is the relation to be altered.
   b) *A* is the attribute to be added and *D* is its domain.
2) To remove an attribute: **alter table** *r* **drop** *A*.
   a) *r* is the relation to be altered.
   b) *A* is the attribute to be deleted from the relation.

3 Basic Structure of SQL Queries

Consists of three clauses: **select, from, where**.

1) Input relations specified in **from** clause.
2) Operations on input relations defined in **select** and **where** clauses.
3) Basic syntax:

```
select attr_1, attr_2, ..., attr_n
from table_1, table_2, ..., table_m
where P;
```

3.1 Queries on Relations

1) To specify attributes in the **select** clause, one can prepend the attribute name with the relation name separated by a dot (.).
2) In practice, removing duplicates is time-consuming. The default behaviour in SQL is to retain duplicates.
3) To force elimination of duplicates, **distinct** keyword can be used in the **select** clause. To explicitly specify that duplicates should be retained, **all** is used in the **select** clause.
4) The resulting select clause looks like **select (distinct—all)**.
5) The **select** clause can also contain arithmetic expressions on attributes or constants. These queries do NOT modify the underlying relation.
6) The **where** clause selects those tuples that satisfy the given predicate. One can join multiple predicates using **and, or** and **not**.
7) The general menaing/result of an SQL query is as follows:
   a) Generate a cartesian product of the relations in the **from** clause.
   b) Apply the predicates in the **where** clause.
   c) Output the attributes of each tuple specified in the **select** clause.

4 Additional Basic Operations

4.1 The Rename Operation

Names cannot always be derived from a relation because:

1) Two relations in the **from** clause may have the same attribute names.
2) Arithmetic expressions do not even have a name.
3) One may want to change the attribute name even if it is derivable to something more convenient for future use.

To rename attributes, the **as** clause is used.

1) Syntax: *r* **as** *s*.
2) It can appear in both **select** and **from** clauses (to rename relations for convenience).
3) A useful case for **as** in the **from** clause is when a relation is compared to itself, in which case the relation is given two names.
4) The identifiers used to rename relations are called **correlation names** in the SQL standard. Other names are **table alias, correlation variable, tuple variable**.

4.2 String Operations

1) Strings in SQL are enclosed in single or double quotes (‘’ or “”).
2) If a string contains one quote, we can use the other quote to enclose it.
3) Functions on strings in SQL:

a) *Concatenation* using ||.
b) Convert to uppercase using **upper**(s).
c) Convert to lowercase using **lower**(s).
d) Remove spaces at the end of the string using **trim**(s).

4) Pattern matching in SQL using the **like** or **not like** operator. Special characters used are:
   a) Percent (%) to match any substring.
   b) Underscore (_) to match any character.
5) Special characters can be escaped for literal usage, such as \%, \\, and so on.

4.3 Attribute Specification in the Select Clause

1) An asterisk (*) can be used int the **select** clause to denote *all attributes*.
2) For all attributes in a certain table, use `tableName.*`. For all attributes in *all* tables, simply use *.

4.4 Ordering the Display of Tuples

1) **order by** clause in SQL allows the ordering of tuples in a relation.
2) Syntax (in **where** clause): ... **order by** A_1**,** A_2**,** ...
3) Default behaviour is ascending order. To specify the sort order per attribute, we use **order by** A_1 **(asc—desc),** A_2 **(asc—desc), ...**

4.5 Where-Clause Predicates

1) For simplifying **where** clauses, SQL includes a **(not) between** clause.
2) Syntax (in **where** clause): ... **(not) between** v_1 **and** v_2 ...
3) Note that both end values are *inclusive* for this clause.
4) Multiple equalities can be combined by using a *row constructor* () instead of using **and**.

5 Set Operations

SQL supports three set operations: **union, intersect** and **except**.

5.1 The Union Operation

1) Syntax: r_1 **union** r_2, where
   a) **union** denotes the union operation.
   b) $r1$ and $r2$ are compatible relations or subqueries yielding compatible relations.
2) The **union** operation eliminates duplicates unlike the **select** clause. To retain duplicates, use **union all**.

5.2 The Intersect Operation

1) Syntax: r_1 **intersect** r_2, where
   a) **intersect** denotes the except operation.
   b) $r1$ and $r2$ are compatible relations or subqueries yielding compatible relations.
2) The **intersect** operation eliminates duplicates unlike the **select** clause. To retain duplicates, use **intersect all**.

5.3 The Except Operation

1) Syntax: r_1 **except** r_2, where
   a) **except** denotes the except operation.
   b) $r1$ and $r2$ are compatible relations or subqueries yielding compatible relations.
2) The **except** operation eliminates duplicates unlike the **select** clause. To retain duplicates, use **except all**.

6 Null Values

1) Presents special problems while evaluating arithmetic operations and predicates.
2) The result of an arithmetic operation involving **null** is *always* **null**.
3) The result of a comparison involving **null** is *always* **unknown**.
4) Behaviour of **unknown** (T statnds for True, F for False, and U for unknown):

v_1	v_2	**and**	**or**	**not**
T	U	U	T	F
F	U	F	U	T
U	U	U	U	U

5) If a **where** clause evaluates to **false** or **unknown** for a tuple, that tuple is not included in the resulting relation.
6) To test for a **null** value, SQL includes the **is (not) null** predicate. Similarly for **is (not) unknown**.

7 Aggregate Functions

Aggregate Functions: Functions that take a collection of values as input and return a single value. SQL has five standard built-in aggregate functions:

Aggregate Function	SQL Function
Average	**avg**
Minimum	**min**
Maximum	**max**
Total	**sum**
Count	**count**

7.1 Basic Aggregation

1) The result of applying a simple aggregate function is a single attribute containing a single aggregate value.
2) We can rename the atribute to something more convenient using the **as** clause.
3) Usage of **distinct**:
   a) Eliminate duplicates in aggregate functions. Use **distinct** in the aggregate expression.
   b) The use of **distinct** with **count**(*) is **not** allowed.
   c) Using **distinct** with **max** and **min** is legal, but makes no difference.
   d) To specify duplicate retention, we use **all**, but it is the default behaviour.

7.2 Aggregation with Grouping

1) To group sets of tuples for applying aggregates, we use **group by** clause. Tuples with the same attribute on all values in the **group by** clause are placed in one group.
2) When **group by** is used, attributes that appear in the **select** clause without being aggregated MUST appear in the **group by** clause.

7.3 The Having Clause

1) Applies to groups constructed by the **group by** clause.
2) Comes after **group by** clause in a query.
3) Like the **select** clause, any attribute that is present in the **having** clause without aggregation MUST appear in the **group by** clause.
4) For a query containing aggregation, following are the rules to evaluate its result:
   a) Evaluate the **from** clause to get a relation.
   b) Apply the predicate of the **where** clause.
   c) Group these tuples using the **group by** clause.
   d) Apply the **having** clause to each group.
   e) Project selected attributes mentioned in the **select** clause.

7.4 Aggregation with Null and Boolean Values

1) All aggregate functions except **count**(*) *ignore* null values in their input collection.
2) The **count** of an enpty collection is defined to be 0. Other aggregate operations return **null** on an empty collection.
3) For **boolean** data types, **some** and **every** can be used to compute the disjunction (**or**) conjunction (**and**) of the values.

8 Nested Subqueries

A **subquery** is a **select-from-where** expression nested inside another SQL query. Use cases of subqueries are illustrated below.

8.1 Set Membership

1) **(not) in** connective tests for set membership.
2) Can also use these operators in enumerated sets, such as `(val1, val2, ...)`.
3) Use row constructors to test set membership in another relation (if supported by the database system).

8.2 Set Comparison

1) To check if an element returns **true** on comparison with at least one element of another relation, use **(op) some**, where **op** is a comparison operator.
2) Note that =**some** is *identical* to **in**, while ¡¿**some** is *not identical* to **not in**.
3) To check if an element returns **true** on comparison with all elements of another relation, use **(op) all**, where **op** is the comparison operator.
4) Note that ¡¿**some** is *identical* to **not in**, while =**all** is *not identical* to **in**.

8.3 Test for Empty Relations

SQL provides an **exists** construct that returns **true** if the argument relation (subquery) is nonempty. A **not exists** construct is also provided that mirrors this functionality.

8.4 Test for the Absence of Duplicate Tuples

SQL provides a **unique** construct for testing whether the argument subquery has no duplicate tuples. A **not unique** construct that mirrors this action is also provided.

8.5 Subqueries in the From Clause

1) A **select-from-where** query returns a relation, and hence can be nested in the **from** clause of another **select-from-where** query.
2) Such subqueries can also be given names using the **as** constructor (depends on SQL implementation).
3) Nested subqueries in the **from** clause cannot use correlation variables from other relations in the **from** clause.
4) Since SQL:2003 a subquery in the **from** clause that is prefixed by **lateral** can access other correlation variables.

8.6 The With Clause

1) The **with** clause provides a way to define a temporary relation whose definition is available only to the query in which the **with** clause occurs.
2) Introduced in SQL:1999.

8.7 Scalar Subqueries

1) A subquery that returns only one tuple containing a single attribute is called a **scalar subquery**.
2) Can occur in **select, where** and **having** clauses.

8.8 Scalar Without a From Clause

1) Certain queries may have subqueries that use a **from** clause but the top-level query does not.
2) This may lead to an error in some systems. In such cases a *dummy* relation is provided to act as a *placeholder* in the **from** clause of the top level query, such as *dual* in Oracle.

9 Modification of the Database

9.1 Deletion

1) SQL syntax:

```
delete from $r$
where $P$;
```

2) If the **where** clause is omitted, *all* tuples in the argument relation are deleted.
3) The **delete** command operates on *only* one relation.
4) ALL tuples in the relation are *tested* before deletion.

9.2 Insertion

1) To insert into a relation, a single tuple or a relation constaining a set of tuples is specified.
2) SQL syntaxes:

```
insert into tableName(Schema) values (
    v_1, v_2, ..., v_n);
insert into tableName(Schema) <
    subquery>;
```

Note: The subquery must be chosen carefully. For example, queries like

```
insert into r select * from r;
```

can insert an infinite number of tuples into r if r does not have a primary key constraint.

3) One can also insert tuples containing **null** values, provided the attribute is declared in the realtion schema as **not null**.

Updates

- The **update** statement is used to modify $_{s}ome_{v}alue sof the tuples. - SQL syntax$:

¡div class=”syntax” style=”–width: 20em;”¿

update r **set** $(A_1, A_2, \ldots, A_n) = (v_1, v_2, \ldots, v_n)$ **where** P;

¡/div¿

where

- r is the relation to be updated. - A is the assignment statement. - P is the predicate based on which the tuples should be updated. - To have multiple updates occur simutaneously without one affecting the other, a **case** construct is provided. The syntax for this is:

¡div class=”syntax” style=”–width: 11em;”¿

update r **set** A = **case** emsp; emsp; **when** P_1 **then** v_1 emsp; emsp; **when** P_2 **then** v_2 emsp; emsp; … emsp; emsp; **when** P_n **then** v_n emsp; emsp; **else** v_0 **end**;

¡/div¿

where

- r is the relation. - A is the new attribute. - P_i are the predicates. - v_i are the values or expressions for A for each case. - We can also use scalar subqueries in **set** clause. The **case** construct permits us to handle nulls if they are present.