SQL

by Terry Sergeant

Contents

1 Introduction

1.1 What is SQL?

SQL (Structured Query Language) is typically pronounced "sequel" or sometimes "ess-que-ell". It is a language for manipulating or retrieving information from a relational database. SQL queries are entered by typing a command.

These notes are a cursory guide to some of the core features of SQL.

1.2 Why should I care?

- SQL has become the lingua franca for RDBMS's.

> *lingua franca* **1:** a common language that consists of Italian mixed with French, Spanish, Greek, and Arabic and is spoken in Mediterranean ports **2:** any of various languages used as common or commercial tongues among peoples of diverse speech.
>
> *Webster's Ninth New Collegiate Dictionary*

- If you learn SQL you will be able to create queries and manipulate databases in virtually any modern RDBMS!
- Read the above statement again.
- Read it one more time.
- Finally, SQL statements can be generated under the control of a program.

1.3 A Brief History

SQL evolved from an IBM project/language called Sequel. It eventually began to be used more widely and in 1986, ANSI/ISO (American national Standards Institute / International Standards Organization) published an SQL standard. Updated standards followed in 1989, 1992, 1999, and 2003.

`Who gives ANSI/ISO the right to say what SQL should look like?`

1.4 Examples and Exercises

The examples and exercises given below assume the existence of three related tables that represent information about books, authors, and publishers. The tables are assumed to be arranged as follows:

- book=(<u>isbn</u>, title, author_id, num_pages, publisher_id, year)
- author=(<u>author_id</u>, firstname, lastname)
- publisher(<u>publisher_id</u>, name, location)

2 SELECT Queries

A SELECT statement is used to retrieve information from a database in the form of a list. Here is the form for the SELECT statement.

```
SELECT field-list
  FROM table-list
  WHERE conditions
  GROUP BY field-name
  HAVING conditions
  ORDER BY field-list
```

It should be noted that:

- SQL is a free format language. So, the entire SELECT statement could be written on a single line or could be spread out over several lines as show below.
- In long SQL statements it can be more readable to spread over several lines.
- Not every SELECT statement uses all of the clauses given above. The simplest statements will use only SELECT and FROM.
- The *field-list* is simply a list of fields you want to see, separated by commas. If you want to see all the fields simply put an asterisk (*).
- The *table-list* is a list of tables from which the query should be drawn. If multiple tables are listed the DBMS will perform a cross-product of the tables listed. **Do you know what a cross-product is?**

Example 1 Here is a simple `SELECT` query that lists all fields in the table `book`.

```
SELECT * FROM book;
```

2.1 The `WHERE` Clause

- The `WHERE` *`conditions`* clause is used to specify the criteria necessary for matching records.
- This clause can be simple or quite complex, but is basically used to restrict the number of responses coming from the query.
- Boolean operator such as `not`, `and`, and `or` can be used to combine conditions.
- String contants require single quotes and the percent symbol (`%`) can be used as a wildcard. Also, the constant `true` and `false` can be used.

Example 2 This `SELECT` query lists all book titles in the table `book` that were published in 1999.

```
SELECT title FROM book
WHERE year=1999;
```

2.2 Some Exercises Using `WHERE`

Try to construct `SELECT` queries that will produce the following results:

1. Display title, pages, and year of all books.
2. Display all fields in the `book` table that have fewer than 100 pages.
3. Display a list of all old (published before 1960), long (at least 525 pages) books.
4. Show all fields in `author` table for authors whose last name is "Ford".
5. Show all authors whose last name starts with "Fo".

2.3 Joining Tables

Up to this point we have not tried to combine information from more than one table into a single query. It's not hard to imagine that you might want a list of books together with the names of the authors of those books. The author names and the book titles are in separate tables so we must join the two tables. There are several ways to achieve this

Example 3 Here we list book titles and author names for all books by performing a Cartesian product and then restricting the results with a `WHERE` clause.

```
SELECT title, firstname, lastname FROM book, author
WHERE book.author_id=author.author_id;
```

NOTE: Observe the dotted notation to specify which `author_id` field we mean. This is necessary because both tables listed in the `FROM` clause have a field named `author_id`. **What would happen if we omit the `WHERE` clause?**

ALSO NOTE: This query produces 1721 rows in our sample data.

We can achieve the same results using an `INNER JOIN` operation as follows:

Example 4 Here we list book titles and author names for all books by using `INNER JOIN`.

```
SELECT title, firstname, lastname
FROM book INNER JOIN author ON book.author_id=author.author_id
```

When using a `JOIN` the conditions for joining the tables are given in the `FROM` clause rather than the `WHERE` clause.

NOTE: This query also produces 1721 rows in our sample data.

The term `INNER JOIN` seems to imply the existence of an "outer join" as well. It turns out that there are *three* types of outer joins: `FULL OUTER JOIN`, `LEFT OUTER JOIN`, and `RIGHT OUTER JOIN`. Up to this point we have been operating under the implicit assumption that every book has a listed author and every author in the database has a book. In some causes this may not be a good assumption. **Why not?**

The `INNER JOIN` only shows query results for which that implicit assumption is correct. Consider and view the results of the following example:

Example 5 Here we list book titles and author names for all books by using `FULL OUTER JOIN`.

```
SELECT title, firstname, lastname
FROM book FULL OUTER JOIN author ON book.author_id=author.autho
```

NOTE: This list has a significant number of `NULL` book titles and a whopping 6022 rows in our sample data!

The reason for this is that there are, apparently, a large number of authors in the database who have no books in the `book` table. The distinction between `LEFT OUTER JOIN` and `RIGHT OUTER JOIN` are simply a matter of whether we want to include entries with no matching values in the first listed table (i.e., the one on the left ... in this case `book`) or we want to include entries with no matching values in the second listed table (i.e., the one on the right ... in this case `author`).

In this particular data set a `LEFT OUTER JOIN` produces the same results as an `INNER JOIN` because there are no books without an author in the `book` table. A `RIGHT OUTER JOIN` in this set produces the same results as the `FULL OUTER JOIN` because there are many authors without books in the `book` table.

The requirement of having to specify which field you want a `JOIN` operation can be tedious. This tedium can be alleviated by adding the keyword `NATURAL` in front of the join operator. A "natural" join instructs the SQL engine to join the tables on the columns whose names match. Consider this example:

Example 6 Here we list book titles and author names for all books by using `NATURAL INNER JOIN`.

```
SELECT title, firstname, lastname
FROM book NATURAL INNER JOIN author;
```

NOTE: This list is united on the `author_id` since both `book` and `author` contain a field by that name.

The `NATURAL` modifier can be used with outer joins as well.

2.4 Some Exercises Using JOIN

Try to construct SELECT queries that will produce the following results:

1. Display title, pages, year, author first and last name of all books.
2. Display all fields in the book and publisher tables that have fewer than 100 pages.
3. Display a list of all old (published before 1960), long (at least 525 pages) books. Include author name and publisher name in the list.
4. Show all authors that don't have a book associated with them.
5. Experiment with inner, left outer, right outer, and full outer and observe how the counts of queries change.

2.5 Aggregate Functions, GROUP BY, and HAVING

2.5.1 Aggregate Functions

- Aggregate functions can be used to apply some common mathematical operations to a database column.
- Some aggregate functions include COUNT, SUM, MAX, MIN, AVG.

Example 7 This SELECT query uses an aggregate function to count the number of books in the book table.

```
SELECT COUNT(title) FROM book;
```

Example 8 This SELECT query uses an aggregate function to determine the age of the oldest book in the book table.

```
SELECT MIN(year) FROM book;
```

2.5.2 The GROUP BY Clause

This clause is often used in conjuction with aggregate functions to group entries into subsets.

Example 9 This SELECT query uses GROUP BY to list show the number of books for each publisher.

```
SELECT publisher_id, COUNT(title) FROM book
GROUP BY publisher_id;
```

2.5.3 The HAVING Clause

This HAVING clause serves the same basic purpose as the WHERE clause, except that it can be applied to grouped results. Suppose, for example, we wanted to show the number of books for each publisher, but only for those who have more than 50 books to their credit.

Example 10 This query uses GROUP BY, and HAVING to list show the number of books for each publisher who has at least 50 books.

```
SELECT publisher_id, COUNT(title) FROM book
GROUP BY publisher_id
HAVING COUNT(title) >= 50;
```

2.6 The ORDER BY Clause

- This clause provides a simple way to organize the result of any query by sorting them on one or more fields.
- When sorting on multiple fields, the DBMS uses a stable sorting routine so that secondary fields are sorted within "equal" values of the primary sort field.
- The key word DESC can be used to reverse the default order of sorting.
- The ORDER BY clause can be applied to aggregate functions as well.

Example 11 List all books from oldest to newest. Books in the same year should be ordered alphabetically by title.

```
SELECT * FROM book
ORDER BY year, title;
```

Example 12 Show average book length for book published in the same year. List those years with the longest averages first.

```
SELECT year, AVG(num_pages) FROM book
GROUP BY year
ORDER BY AVG(num_pages) DESC;
```

2.7 The SANTA Clause

In case you haven't heard this from anyone before, let me be the first to say: "There is no such thing as SANTA Clause."

There are, however, a few "extra" features that would be nice to know. When selecting a field you can rename it to make it more descriptive. For example:

Example 13 Show average book length for book published in the same year and given meaningful column names.

```
SELECT year AS "Publication Year",
       AVG(num_pages) AS "Average Book Length"
FROM book
GROUP BY year;
```

Also, when checking to see if a field is null you must use the keyword IS rather than the equal sign as follows:

Example 14 List authors with no first name.

```
SELECT * FROM author
WHERE firstname IS NULL;
```

Finally, when using wildcards (% to match multiple characters or _ to match a single character) you must use the keyword LIKE rather than the equal sign as follows:

Example 15 List all titles that begin with the word "The".

```
SELECT * FROM book
WHERE title LIKE 'The%'
```

2.8 Some Exercises Using `GROUP BY`, `HAVING`, and `ORDER BY`

Try to construct `SELECT` queries that will produce the following results:

1. Produce a list of book titles and publisher ids sorted by publisher id and then by title.
2. Display the length of the longest book for each publisher id; sort the results in descending order of book length.

2.9 Creating Subqueries

Some queries require as conditions, results from other queries. You can write a query with a query inside of it.

Example 16 Write a query that will list all books that have at least as many pages as the longest book written in the 1970's. First it would be useful to find out how long the longest book in the 1970's is. We might accomplish that result like this:

```
SELECT MAX(num_pages) FROM book
WHERE year>=1970 and year<1980;
```

Execution of this query on our sample data produces the value 550. With that value in hand we can construct the following query to obtain the answer we're looking for:

```
SELECT * FROM book
WHERE num_pages >= 550;
```

Of course, it is not very satisfying to write two separate queries and to have to remember the result of one and hard-code that value in the second. One solution is write nest queries as follows:

```
SELECT * FROM book
WHERE num_pages >= (SELECT MAX(num_pages) FROM book
                    WHERE year>=1970 and year<1980);
```

It should be noted again, at this point, that SQL is a free-format language. So, the indentation in this example is not required, but simply improves readability.

2.10 Creating and Using Views

A little bit of imagination should convince you that it can become difficult to construct a correct query when queries are nested within queries 5 levels deep. SQL allows you to name a query. A named query is called a *view*. Once a view is created it can be invoked by name as if it is a table. The view can be used as part of other queries. This allows abstraction of possibly complicated results to be used as building blocks for even more complicated results.

Example 17 Consider the previous example of writing a query that will list all books that have at least as many pages as the longest book written in the 1970's. First we can create a view named `longestof70s`. NOTE: This step of creating the view needs only to be done once. Once the view is created it can be invoked using it's name.

```
CREATE VIEW longestof70s AS
  SELECT MAX(num_pages) FROM book
  WHERE year>=1970 and year<1980;
```

Now the final query can be simplified as follows:

```
SELECT * FROM book
WHERE num_pages >= (select max from longestof70s);
```

It is common the create views for any queries that will be used repeatedly.

2.11 Some Exercises Using Subqueries and Views

1. Display all books longer than the longest written by the author whose id number is 4159.
2. Create a view called `prolific_authors` that lists the id numbers and number of books of all authors who have written more than 5 books listed in the database.
3. Using the view just created write a query that will list the names of all the prolific authors. HINT: The `WHERE` clause can use the `IN` operator to determine membership in a list. (e.g., `WHERE id IN (some list of id numbers)`)

2.12 Some More Exercises

1. Show all fields of books and authors for those books whose authors have no first name;
2. Count the number of authors who have no books in the `book` table.
3. Create a view called `allinfo` that shows all fields of the `book`, `author`, and `publisher` tables joined (inner) on the appropriate fields; the results should be ordered by author (lastname then firstname) and then by title.
4. Use the `allinfo` view as a basis for listing all fields of that view for books published by `American Publishing Co.` (whose id is 5).
5. Show the length of the longest book for each publishing company together with publisher id, company name and location; sort results in order of book length (longest books first).
6. List all publishers (in descending order of count) who have published at least 25 books before 1960.
7. List authors (along with their average book length) whose average book length is at least 450 pages; order by author name.
8. Repeat previous query but only include authors who have authored more than one book.

3 `DELETE`, `INSERT`, and `UPDATE`

All of the queries we've done so far simply allow us to extract lists from the existing database. SQL also provides commands for modifying the content of the database.

3.1 DELETE Queries

The SQL command to remove rows from a table has this form:

```
DELETE FROM table
WHERE conditions
```

WARNING: This command can delete many rows depending on the criteria specified:

Example 18 Delete all books published by American Publishing Company.

```
DELETE FROM book
WHERE publisher_id=5;
```

Example 19 Delete all books whose author doesn't have a first name.

```
DELETE FROM book
WHERE author_id IN
  (SELECT author_id FROM author
   WHERE lastname IS NULL);
```

IMPORTANT: It is highly recommended that you construct a `SELECT` query to identify the entries you want to remove. Once you are certain you are selecting the correct group of records, then modify the `SELECT` query to become a `DELETE` query.

3.2 INSERT Queries

`INSERT` queries allow you to add a row to a database table. The basic form for `INSERT` is this:

```
INSERT INTO table (field-1, field-2, ..., field-n)
VALUES (value-1, value-2, ..., value-n)
```

Example 20 Insert a new publisher into the `publisher` table.

```
INSERT INTO publisher (publisher_id,name,location)
VALUES (15,'Gorgonzola Publishing Co.','Naples, Italy');
```

NOTE: This particular table does not perform autonumbering of id numbers for new entries. Thus, we had to specify an id number that did not conflict with existing id numbers. If the `publisher_id` field *was* an autonumbering field then the word `default` could have been used in place of the number 15.

ALSO NOTE: The order of fields listed after the table name must match the order given after the `VALUES` clause. The field order does not, however, have to match the order in which the fields appear in the database. If the fields *do* appear in the order saved in the database then the list of fields can be omitted.

Example 21 Insert a new publisher into the `publisher` table but this time we assume that `publisher_id` is an autonumber field and that we know the order of fields as they are stored in the database.

```
INSERT INTO publisher
VALUES (default,'Gorgonzola Publishing Co.','Naples, Italy');
```

3.3 UPDATE Queries

The basic form of an `UPDATE` query is as follows:

```
UPDATE name-of-table
  SET column1-name=expression1,
      column2-name=expression2,
        .             .
        .             .
        .             .
  WHERE condition
```

Example 22 It should be noted that the UPDATE can apply to a single row as in this example.

```
UPDATE author SET firstname='Iam',lastname='Windy'
WHERE author_id=7
```

Example 23 Alternatively, multiple rows can be targeted by having a more inclusive `WHERE` clause. WARNING: If we omit the `WHERE` clause altogether the query will affect all rows in the table!

```
UPDATE books SET author_id=7 WHERE num_pages > 400
```

4 CREATE and DROP Queries

In section 3, the queries examined were used to modify the contents of existing tables. The `CREATE` and `DROP` queries allow the creation new tables and deletion of tables, respectively.

4.1 CREATE Queries

The basic form of a `CREATE` query is as follows:

```
CREATE TABLE name-of-table (
        field-1  type-of-field-1 [modifiers],
        field-2  type-of-field-2 [modifiers],
           .          .             .
           .          .             .
           .          .             .
        field-n  type-of-field-n [modifiers],
        [integrity-contraints]
);
```

Consider one possible way to create the `book` table we've been using in our examples:

Example 24 Create a `book` table with fields named `isbn`, `title`, `author_id`, `num_pages`, `publisher_id`, and `year`. Also mark `isbn` as the primary key.

```
CREATE TABLE book(
        isbn            CHAR(11),
        title           VARCHAR(80) NOT NULL,
        author_id       INT NOT NULL,
        num_pages       SMALLINT,
        publisher_id    INT,
        year            CHAR(4),
        PRIMARY KEY (isbn)
);
```

The appearance of NOT NULL following the type for title and for author_id instruct the DBMS to prevent a row from being added unless values for these fields are provided. Another common modifier is UNIQUE which requires that no other row contain a matching value for this field.

The final statement PRIMARY KEY(isbn) is an integrity contraint that marks the isbn field as the primary key. This designation imposes the restrictions that the designated field cannot be null and must be unique.

NOTE: There are several integrity contraints besides PRIMARY KEY that we'll learn at a later time.

ANOTHER NOTE: Rules regarding table names are DBMS-dependent. In PostgreSQL named can be composed of letters, digits, and underscore symbols. I personally recommend that you use only lowercase letters in PostgreSQL because it requires that you put quotation marks around every field and table name containing uppercase letters.

There are also a variety of field types available. Most of the types described in the table below are standard SQL types with the notable exception of the SERIAL type.

Type	**Description**
CHAR(n)	exactly *n* characters
VARCHAR(n)	up to *n* characters
INT	integers (range is specified by DBMS)
SMALLINT	integers but with restricted range
NUMERIC(p,d)	a fixed point number with *p* digits and *d* decimals
REAL	floating point numbers (range specified by DBMS)
FLOAT(n)	floating point numbers with up to *n* digits
DATE	year, month, date
TIME	time of date with hours, minutes, seconds
TIMESTAMP	date and time together
SERIAL	PostgreSQL type for autonumber field; the underlying type is INT with a default modifier that makes use of the PostgreSQL nextval function

4.2 DROP Queries

Removing table is, obviously much simpler than creating them. The form of the DROP query is this:

```
DROP TABLE name-of-table;
```

As with the CREATE command, the syntax implies that we might be able to "drop" elements besides tables (e.g., views). Some tables have integrity contraints that may require other tables to be dropped first.