Introduction to SQL

Chapter 4

Michael Tsiang

Stats 167: Introduction to Databases

UCLA

Getting Started

The Structure of SQL Databases

Types of SQL Commands

Basic SQL Queries

Leveraging Intuition From R

Getting Started

What is SQL?

A **database** is an organized collection of data.

A **relational (or SQL) database** is a database whose data is split into multiple tables, each of which stores related data.

Structured Query Language (SQL) is a language designed for communicating with relational databases.

Nearly every relational database management system (RDBMS or just DBMS) supports SQL, so learning one language will allow you to interact with nearly any database you will encounter.

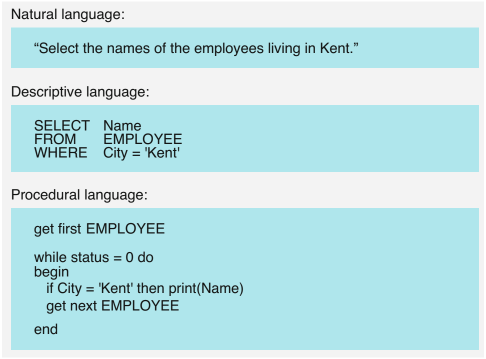
While learning a new language may seem daunting, SQL has fairly limited vocabulary and minimal syntax rules, so learning to read and write common SQL statements is largely intuitive.

Side Note: SQL is sometimes pronounced as the letters "S-Q-L" or as "sequel."

SQL is Descriptive

The ease of readability of SQL is because it is a **descriptive** language, which means SQL commands describe the actions we want and the DBMS will determine how to perform the actions.

In contrast, a **procedural** language would require specifying the method of how to perform the actions.



Natural language:

"Select the names of the employees living in Kent."

Descriptive language:

```
SELECT   Name
FROM     EMPLOYEE
WHERE    City = 'Kent'
```

Procedural language:

```
get first EMPLOYEE

while status = 0 do
begin
  if City = 'Kent' then print(Name)
  get next EMPLOYEE
end
```

Source: Figure 1.5 in SQL and NoSQL Databases by Kaufmann and Meier, Second Edition, Springer, 2023.

SQL Extensions

Standard SQL is governed by the American National Standards Institute (ANSI) standards committee, so it is sometimes called **ANSI SQL**.

Many DBMSs have extended ANSI SQL by adding statements to provide additional functionality or simplification for certain operations. Individual implementations of SQL have their own names (e.g., PL-SQL is used by Oracle, Transact-SQL is used by Microsoft SQL Server, etc.).

We will focus on ANSI SQL (since it is shared by all DBMSs), but we will note if/when we use DBMS-specific SQL statements.

Common DBMSs

The specific DBMS you may use in the future will depend on the company you work for (and possibly even the operating system you use). Some common DBMSs are Microsoft SQL Server, MySQL, and PostgreSQL.

We will use **SQLite**, which is a serverless DBMS that does not require any complicated installation or configuration.

For more information on SQLite: https://www.sqlite.org/index.html

DB Browser for SQLite

For a free downloadable software to use SQLite, we will use **DB Browser for SQLite (DB4S)**.

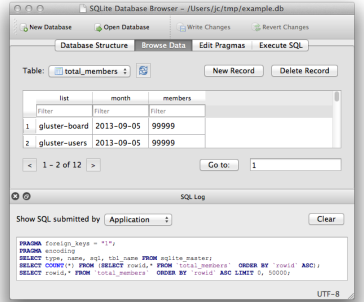


Table: total_members

	list	month	members
	Filter	Filter	Filter
1	gluster-board	2013-09-05	99999
2	gluster-users	2013-09-05	99999

1 - 2 of 12

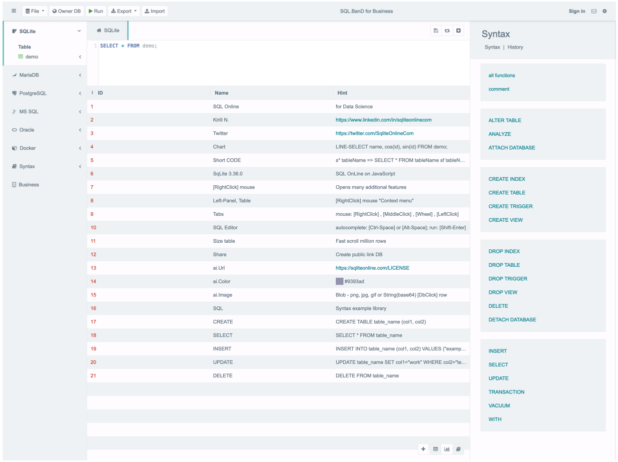
Show SQL submitted by Application

```
PRAGMA foreign_keys = "1";
PRAGMA encoding
SELECT type, name, sql, tbl_name FROM sqlite_master;
SELECT COUNT(*) FROM (SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC);
SELECT rowid,* FROM `total_members`  ORDER BY `rowid` ASC LIMIT 0, 50000;
```

Download at: https://sqlitebrowser.org/dl/

SQLite Online

For a web-based alternative (no download required) to implement SQLite in any web browser: https://sqliteonline.com/



```
SELECT * FROM demo;
```

ID	Name	Hint
1	SQL Online	for Data Science
2	Kirill N.	https://www.linkedin.com/in/sqliteonlinecom
3	Twitter	https://twitter.com/SqliteOnlineCom
4	Chart	LINE-SELECT name, cos(id), sin(id) FROM demo;
5	Short CODE	s* tableName => SELECT * FROM tableName sf tableN…
6	SqLite 3.36.0	SQL OnLine on JavaScript
7	[RightClick] mouse	Opens many additional features
8	Left-Panel, Table	[RightClick] mouse "Context menu"
9	Tabs	mouse: [RightClick] , [MiddleClick] , [Wheel] , [LeftClick]
10	SQL Editor	autocomplete: [Ctrl-Space] or [Alt-Space]; run: [Shift-Enter]
11	Size table	Fast scroll million rows
12	Share	Create public link DB
13	ai.Url	https://sqliteonline.com/LICENSE
14	ai.Color	▨ #9393ad
15	ai.Image	Blob - png, jpg, gif or String(base64) [DbClick] row
16	SQL	Syntax example library
17	CREATE	CREATE TABLE table_name (col1, col2)
18	SELECT	SELECT * FROM table_name
19	INSERT	INSERT INTO table_name (col1, col2) VALUES ("examp…
20	UPDATE	UPDATE table_name SET col1="work" WHERE col2="te…
21	DELETE	DELETE FROM table_name

Accessing the Data

To access a database in most DBMSs, you will need to establish a connection to a server with login credentials (usually provided by your employer).

For SQLite, data can be loaded using a `.sql` or `.sqlite` file.

The data used in this lecture is in the **`TYSQL.sqlite`** file posted on the course website.

The data is from the book *Sams Teach Yourself SQL in 10 Minutes a Day, 5th Edition* by Ben Forta.

Book website: https://forta.com/books/0135182794/

Loading the Data

In DB Browser for SQLite:

- Click on “Open Database”
- Find the folder that contains the `.sqlite` data file, click on the file name, and then click “Open”.

The “Execute SQL” tab will open the window to write and execute SQL statements.

In SQLite Online:

- Click on “+”
- Click on “Open SQLite DB”
- Find the folder that contains the `.sqlite` data file, click on the file name, and then click “Open”.

The main window on SQLite Online has the window to execute SQL statements.

The Structure of SQL Databases

The Structure of SQL Databases

Data in a relational database is typically organized into **tables** of data, where each **record** (or observation) is stored in **rows** and each **field** (or attribute or variable) is stored in **columns**.

Each column in a database has an associated **datatype**, which defines the type of data the column can contain, e.g., dates, text, currency, integers, etc. The datatype of a column restricts the type of data that can be stored in it.

Tables in a database are organized similar to tidy data frames in R:

- Every row is an observation.
- Every column is a variable.
- Every column has a specific data type.

Primary Keys

A minimal set of columns (or attributes) whose values uniquely identifies every row in a table is called a **primary key**.

Any column in a table can be defined as the primary key, as long as it satisfies the following conditions:

- No two rows have the same primary key value.
- Every row must have a value in the primary key column(s) (i.e., no `NULL` values).
- Primary key values should never be modified or updated.
- Primary key values should never be reused.

Examples: Student ID numbers, ISBN for books, driver's license numbers, etc.

Note: Primary keys are not technically required, but they are usually seen as necessary, as they are fundamental to creating relations between tables.

Foreign Keys

The relations between tables in a relational database are specified through foreign keys.

A **foreign key** is a column in a table whose values must be listed in a primary key of another table.

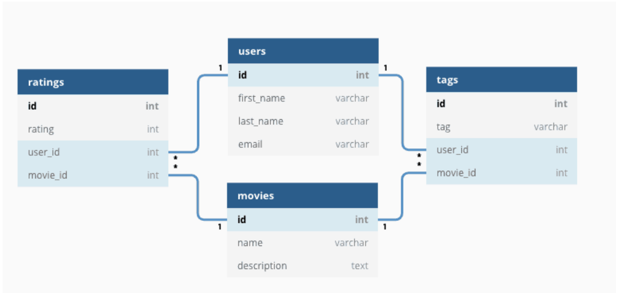
Foreign keys are used to create references between tables.

They also ensure **referential integrity**, so no accidental updates or deletions of rows in one table affects other tables.

Schemas

The **schema** of a database is an outline of its overall structure.

A schema for a relational database about movie ratings is shown below[1].



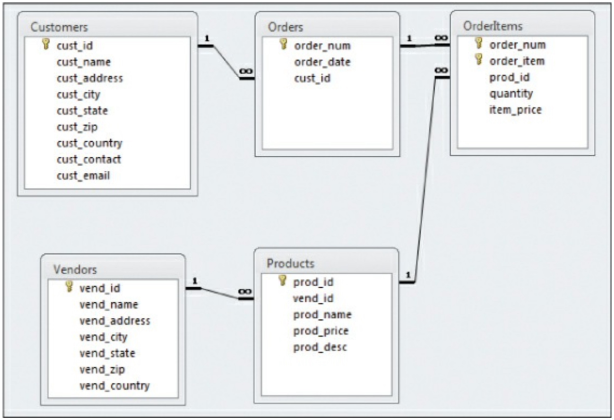
The `id` key in the `users` table relates the `users` data to the `ratings` and `tags` tables as `user_id`, while the `id` key in the `movies` table relates to `ratings` and `tags` as `movie_id`.

Notice that the `users` and `movies` tables do not have a direct relationship, rather they are connected through the other tables in the database.

[1]Source: https://www.heavy.ai/technical-glossary/relational-database

Schema of the TYSQL Data

The TYSQL database we will use in this lecture contains data from a fictitious toy store. This is a schema of the TYSQL database:



Source: Ben Forta, *SQL in 10 Minutes a Day, Sams Teach Yourself, 5th Edition*, Pearson, 2020.

Browse the Data

Before we introduce SQL statements, you can browse through the `TYSQL` database first to get a sense of the data.

In DB Browser for SQLite:

- Click on the “Browse Data” tab
- Click on the Table you want to browse

In SQLite Online:

- In the left window with the database information, right-click on the table name
- Click on “SELECT (Show table)”

Types of SQL Commands

SQL Sublanguages

The SQL language can be separated into a few categories of commands, or sublanguages, that perform various functions.

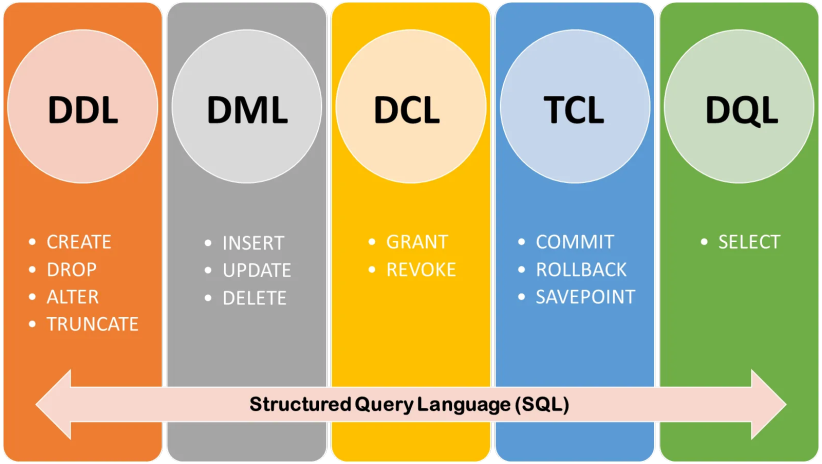
The main categories of commands are:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)
- Transaction Control Language (TCL)
- Data Query Language (DQL)

We will not emphasize most of these sublanguages, but it is helpful to have an overview of the language and have a basic understanding of their purpose.

Types of SQL Commands

An overview of the SQL sublanguage categories:



Source: Analytics Vidhya article “A Quick Refresher on All the Commonly used SQL Commands!”

Data Definition Language (DDL)

The **Data Definition Language (DDL)** is the part of SQL used to create and restructure database objects, such as creating or deleting (or dropping) a table.

The fundamental DDL commands are:

- `CREATE`
- `ALTER`
- `DROP`
- `TRUNCATE`

Data Manipulation Language (DML)

The **Data Manipulation Language (DML)** is the part of SQL used to manipulate data within objects of a database.

The fundamental DML commands are:

- `INSERT`
- `UPDATE`
- `DELETE`

Data Control Language (DCL)

The **Data Control Language (DCL)** is the part of SQL used to control database access privileges.

The fundamental DCL commands are:

- `GRANT`
- `REVOKE`

Transaction Control Language (TCL)

A **transaction** is a unit of work that consists of one or more database operations. By using and managing transactions, the DBMS ensures that CRUD operations (e.g., updating entire rows of data) are done either wholly completed or not at all.

The **Transaction Control Language (TCL)** is the part of SQL used to manage database transactions.

The fundamental TCL commands are:

- `COMMIT`
- `ROLLBACK`
- `SAVEPOINT`

Data Query Language (DQL)

The **Data Query Language (DQL)** is the part of SQL used to query or retrieve data from a database.

The single fundamental DQL command is:

- `SELECT`

Even though the DQL is ultimately based on a single command, it has additional clauses that are added to specify the exact data we want to query.

The DQL will be the primary focus of the course, as it encompasses by far the most common SQL commands you will use as a data scientist.

Basic SQL Queries

Basic SQL Statement Terminology

Some basic terminology:

- A SQL command is called a **statement**.
- SQL statements are made up of one or more **keywords**, which are defined words in the SQL language.
- A SQL statement that returns a dataset is called a **query** (informally, statement and query are often used interchangeably).
- A component of a SQL statement is called a **clause**, usually consisting of a keyword and data.

Basic SQL Syntax Rules

Some general SQL syntax rules:

- Keywords are **not case sensitive**, though tables, columns, and values might be.

  We will write keywords in ALL CAPS for clarity.

- Multiple SQL statements must be separated by semicolons (`;`), but a semicolon is not always required for single SQL statements (it can depend on the DBMS).

  We will include a semicolon at the end of each statement to ensure they always work.

- Extra whitespace in SQL statements is ignored, but it is common to break up statements into multiple lines for readability and debugging.

A Full SQL Query Example

We will start from a full SQL query example to show its readability and then discuss its component parts.

```sql
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2
ORDER BY num_prods;
```

vend_id	num_prods
FNG01	2
BRS01	3

What is this SQL query asking for?

SELECT and FROM

The most common statement in SQL is the **SELECT** statement, which is used to retrieve columns of data from one or more tables.

In its basic form, a SELECT statement needs the column name(s) to select and a **FROM** clause that specifies which table to select from.

For example, the following query selects the prod_name column from the Products table:

```sql
SELECT prod_name
FROM Products;
```

SELECT Output

```sql
SELECT prod_name
FROM Products;
```

prod_name
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
Raggedy Ann
King doll
Queen doll

Selecting Multiple Columns

A single `SELECT` statement can be used to select multiple columns, with column names separated by commas.

```sql
SELECT prod_id, prod_name, prod_price
FROM Products;
```

prod_id	prod_name	prod_price
BR01	8 inch teddy bear	5.99
BR02	12 inch teddy bear	8.99
BR03	18 inch teddy bear	11.99
BNBG01	Fish bean bag toy	3.49
BNBG02	Bird bean bag toy	3.49
BNBG03	Rabbit bean bag toy	3.49
RGAN01	Raggedy Ann	4.99
RYL01	King doll	9.49
RYL02	Queen doll	9.49

Selecting All Columns

The asterisk (∗) **wildcard** character can be used to select all columns from a table.

```sql
SELECT *
FROM Products;
```

prod_id	vend_id	prod_name	prod_price	prod_desc
BR01	BRS01	8 inch teddy bear	5.99	8 inch teddy bear, comes with cap and jacket
BR02	BRS01	12 inch teddy bear	8.99	12 inch teddy bear, comes with cap and jacket
BR03	BRS01	18 inch teddy bear	11.99	18 inch teddy bear, comes with cap and jacket
BNBG01	DLL01	Fish bean bag toy	3.49	Fish bean bag toy, complete with bean bag worms with which to feed it
BNBG02	DLL01	Bird bean bag toy	3.49	Bird bean bag toy, eggs are not included
BNBG03	DLL01	Rabbit bean bag toy	3.49	Rabbit bean bag toy, comes with bean bag carrots
RGAN01	DLL01	Raggedy Ann	4.99	18 inch Raggedy Ann doll
RYL01	FNG01	King doll	9.49	12 inch king doll with royal garments and crown
RYL02	FNG01	Queen doll	9.49	12 inch queen doll with royal garments and crown

SELECT DISTINCT

To select distinct (or unique) rows, include the **`DISTINCT`** keyword in the `SELECT` statement.

```sql
SELECT DISTINCT vend_id, prod_price
FROM Products;
```

vend_id	prod_price
BRS01	5.99
BRS01	8.99
BRS01	11.99
DLL01	3.49
DLL01	4.99
FNG01	9.49

Note: `SELECT DISTINCT` will be applied to all columns, so the query above retrieved all six unique combinations of `vend_id` and `prod_price`.

WHERE

The `WHERE` clause is used to specify **search criteria** or a **filter condition**.

For example:

```sql
SELECT prod_name, prod_price
FROM Products
WHERE prod_price = 3.49;
```

prod_name	prod_price
Fish bean bag toy	3.49
Bird bean bag toy	3.49
Rabbit bean bag toy	3.49

WHERE Clause Operators

The `WHERE` clause supports many conditional operators:

Operator	Description
`=`	Equality
`<>`	Nonequality
`!=`	Nonequality
`<`	Less than
`<=`	Less than or equal to
`!<`	Not less than
`>`	Greater than
`>=`	Greater than or equal to
`!>`	Not greater than
`BETWEEN`	Between two specified values
`IS NULL`	Is a `NULL` (i.e., missing) value

Note: Some operators listed are redundant (e.g., `!<` and `>=` are equivalent), and not all operators are supported by all DBMSs.

Some WHERE Examples

Some queries that use the WHERE clause:

```sql
SELECT vend_id, prod_name
FROM Products
WHERE vend_id <> 'DLL01';
```

```sql
SELECT prod_name, prod_price
FROM Products
WHERE prod_price BETWEEN 5 AND 10;
```

```sql
SELECT cust_name
FROM Customers
WHERE cust_email IS NULL;
```

Note: Strings are usually enclosed in single quotation marks (`''`), though double quotation marks (`""`) also work here.

The AND, OR, and NOT Operators

The **AND**, **OR**, and **NOT** logical operators can be used to combine or negate WHERE clause conditions.

For example:

```sql
SELECT prod_id, prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' AND prod_price <= 4;
```

```sql
SELECT prod_id, prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01';
```

```sql
SELECT prod_name, prod_price
FROM Products
WHERE NOT vend_id = 'DLL01';
```

Use Parentheses

For complicated `WHERE` clauses, use parentheses to clarify the order of operations:

```
SELECT prod_name, prod_price
FROM Products
WHERE (vend_id = 'DLL01' OR vend_id = 'BRS01')
   AND prod_price >= 10;
```

Side Note: In the natural order of operations, `AND` takes precedence over `OR`, but it is better not to rely on remembering the order.

The IN Operator

The IN operator is used to specify a list of valid values, any one of which can be matched. The valid values are enclosed in parentheses and separated by commas.

```sql
SELECT prod_name, prod_price
FROM Products
WHERE vend_id IN ('DLL01', 'BRS01');
```

For short lists, this is equivalent to (though slightly more efficient than) OR:

```sql
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01';
```

Calculated Fields and Aliases

A **calculated field** is a new column that is created by combining values in other columns.

The calculated field does not inherently have a (column) name. The `AS` keyword is used to assign an **alias**, or alternate name, to the column in the output.

The standard arithmetic operations `+`, `-`, `*`, and / work when calculating numeric calculated fields.

Side Note: The `AS` keyword can be used to assign an alias to any column, not just calculated fields.

Example of a Calculated Field

For example:

```
SELECT prod_id,
       quantity,
       item_price,
       quantity * item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

prod_id	quantity	item_price	expanded_price
RGAN01	5	4.99	24.95
BR03	5	11.99	59.95
BNBG01	10	3.49	34.90
BNBG02	10	3.49	34.90
BNBG03	10	3.49	34.90

Aggregate Functions

An **aggregate function** is a function that operates on several rows to calculate and return a single value.

The five aggregate functions in SQL are listed below.

Function	Description
`AVG()`	Returns a column's average (mean) value
`COUNT()`	Returns the number of rows in a column
`MAX()`	Returns a column's maximum value
`MIN()`	Returns a column's minimum value
`SUM()`	Returns the sum of a column's values

Aggregate Functions Examples

For example:

```sql
SELECT AVG(prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

avg_price
3.865

```sql
SELECT SUM(item_price * quantity) AS total_price
FROM OrderItems
WHERE order_num = 20005;
```

total_price
1648

Two Ways to `COUNT()`

The `COUNT()` function can be used in two ways:

- `COUNT(*)` will count the number of rows in a table, whether columns contain values or `NULL` values.
- `COUNT(column)` will count the number of rows that have values in a specific column, ignoring `NULL` values.

Note: All other aggregate functions will automatically ignore column rows with `NULL` values.

COUNT Examples

For example:

```sql
SELECT COUNT(*) AS num_cust
FROM Customers;
```

num_cust
6

```sql
SELECT COUNT(cust_email) AS num_cust
FROM Customers;
```

num_cust
4

GROUP BY

The **`GROUP BY`** clause is used to divide data into logical sets in order to perform aggregate calculations on each group.

For example:

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id;
```

vend_id	num_prods
BRS01	3
DLL01	4
FNG01	2

HAVING

In conjuction with the `GROUP BY` clause, the **`HAVING`** clause is used to filter which groups to include or exclude.

For example:

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id
HAVING COUNT(*) >= 2;
```

cust_id	orders
1000000001	2

WHERE vs HAVING

Both `WHERE` and `HAVING` clauses are used for filtering conditions, and their syntax is identical (other than the keyword).

The main difference between `WHERE` and `HAVING`:

- `WHERE` filters data *before* it is grouped.
- `HAVING` filters data *after* it is grouped.

In particular, when using both clauses, the rows that are eliminated by a `WHERE` clause will not be included in the group, which may change any calculated values and which groups are filtered.

WHERE and HAVING Example

For example:

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

vend_id	num_prods
BRS01	3
FNG01	2

WHERE and HAVING Comparison

Compare the preceding query to the following one:

```sql
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

vend_id	num_prods
BRS01	3
DLL01	4
FNG01	2

In the first query, some rows from the vendor `'DLL01'` were filtered out based on the product price before grouping, even though the vendor has more than 2 products.

ORDER BY

The output of SQL queries is typically displayed in the order it appears in the underlying tables. If entries are deleted or updated, the order may be affected by how the DBMS reuses storage space.

The **`ORDER BY`** clause is used to sort the output by the values in one or more columns. To sort by multiple columns, specify the column names separated by commas.

The default ordering with `ORDER BY` is in ascending order (from A to Z). To sort in descending order, the **`DESC`** keyword can be added after the column name.

ORDER BY Example

For example:

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3
ORDER BY items DESC, order_num;
```

order_num	items
20007	5
20008	5
20006	3
20009	3

Notice that the output is sorted first by `items` (in descending order) and then by `order_num` (in ascending order).

Limiting the Output

SELECT statements return all matched rows. If the output is very large, it may be useful to limit the results to a certain number of rows. Different DBMSs use different keywords to achieve this.

The DBMS-specific keywords used here will not work in other DBMSs.

For example, Microsoft SQL Server uses the **TOP** keyword to limit the number of rows:

```
SELECT TOP 5 prod_name
FROM Products;
```

Variations on Limiting the Output

DB2 uses the **FETCH FIRST** clause, which is specific to that DBMS:

```sql
SELECT prod_name
FROM Products
FETCH FIRST 5 ROWS ONLY;
```

Oracle uses the **ROWNUM** to restrict output based on row numbers:

```sql
SELECT prod_name
FROM Products
WHERE ROWNUM <= 5;
```

LIMIT

In MySQL, MariaDB, PostgreSQL, and SQLite (which we are using), the **LIMIT** clause controls the number of rows to output.

```
SELECT prod_name
FROM Products
LIMIT 5;
```

prod_name
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Fish bean bag toy
Bird bean bag toy

OFFSET

Often used in conjunction with `LIMIT` (though not necessarily), the **`OFFSET`** keyword specifies the number of rows to skip.

For example:

```
SELECT prod_name
FROM Products
LIMIT 5 OFFSET 5;
```

prod_name
Rabbit bean bag toy
Raggedy Ann
King doll
Queen doll

This query returns the 6th through 9th rows of the output (there were only four rows left).

SELECT Clause Ordering

The order of `SELECT` statement clauses matters. The table below shows the order in which the clauses must be used.

Clause	Description	Required
`SELECT`	Columns or expressions to be returned	Yes
`FROM`	Table to retrieve data from	Only if selecting data from a table
`WHERE`	Row-level filtering	No
`GROUP BY`	Group specification	Only if calculating aggregates by group
`HAVING`	Group-level filtering	No
`ORDER BY`	Output sort order	No
`LIMIT`	Limits number of rows	No

Leveraging Intuition From R

It's Beginning To Look A Lot Like `dplyr`

The format and readability of SQL queries is reminiscent of an R command using `dplyr`.

In fact, some of the main clauses in SQL have equivalent functions in `dplyr`, so we can leverage our knowledge of R and `dplyr` to help us gain intuition when reading and writing SQL statements.

`dplyr` Function	SQL Equivalent
`select()`	`SELECT`
`filter()`	`WHERE` or `HAVING` (depends on context)
`group_by()`	`GROUP BY`
`summarize()`	Aggregated columns
`mutate()`	User-defined columns
`arrange()`	`ORDER BY`

Note: The order of `dplyr` functions is not always critical, but SQL is very strict about the order of the clauses in a SQL statement.

Translating SELECT Statements into R

As a simple example, consider the following SQL statement:

```sql
SELECT prod_id, prod_name, prod_price
FROM Products
WHERE prod_price > 8
ORDER BY prod_price;
```

prod_id	prod_name	prod_price
BR02	12 inch teddy bear	8.99
RYL01	King doll	9.49
RYL02	Queen doll	9.49
BR03	18 inch teddy bear	11.99

Question: If `Products` is a data frame (or tibble) in R, how would you translate this SQL query into an R command using `dplyr`?

The dplyr Equivalent

In dplyr:

```
Products |>
   select(prod_id, prod_name, prod_price) |>
   filter(prod_price > 8) |>
   arrange(prod_price)
```

```
  prod_id           prod_name prod_price
1    BR02 12 inch teddy bear       8.99
2   RYL01           King doll       9.49
3   RYL02          Queen doll       9.49
4    BR03 18 inch teddy bear      11.99
```

Note that the components of this dplyr command can be rearranged, but the order of the clauses in the SQL statement matters.

Exercise: Translation to R

Let's revisit this SQL query:

```sql
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
GROUP BY vend_id
HAVING COUNT(*) >= 2
ORDER BY num_prods;
```

vend_id	num_prods
FNG01	2
BRS01	3

Question: Can you translate this SQL query into an R command using `dplyr`?

Solution: The dplyr Equivalent

The same query in dplyr:

```
Products |>
   filter(prod_price >= 4) |>
   group_by(vend_id) |>
   summarize(num_prods = length(vend_id)) |>
   filter(num_prods >= 2) |>
   arrange(num_prods) |>
   select(vend_id, num_prods)
```

```
# A tibble: 2 x 2
  vend_id num_prods
  <chr>       <int>
1 FNG01           2
2 BRS01           3
```