

## 10

### Basic Structured Query Language (SQL)

Structured Query Language (SQL) allows you to retrieve, manipulate, and display information from a database. MS Access lets you perform many of the same tasks using Query Design View, but other database software either lacks this capability or may implement it differently. If you understand the SQL language underlying Access queries, you will be able to formulate queries for essentially any relational database, including ones too large to store in Access. For most of this chapter, we will draw examples from the plumbing store database in Chapter 7, whose design is shown in Figure 10.1.

### Using SQL in Access

You can easily display the SQL form of any query in Access. After pressing the “Query Design” button, you simply select “SQL View” from the “View” button at the left of the “Home” ribbon at the top of the window. When viewing the results of a query, you can also display its SQL form by selecting “SQL View” from the “Home” ribbon. When viewing the SQL form of any query, you may run it by selecting “Datasheet View” on the same “View” button, or by pressing the “!” (Run) button next to it.

### The SELECT ... FROM Statement

The SELECT ... FROM statement is the core of SQL. It specifies which information you want to display. Although SQL has other statements, this book focuses on the SELECT statement.

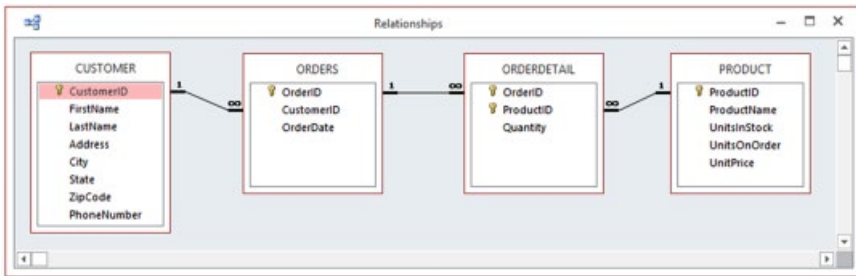
The most basic form for the SELECT statement is:

```
SELECT expressions FROM data_source ;
```

*Introductory Relational Database Design for Business, with Microsoft Access*, First Edition.

Jonathan Eckstein and Bonnie R. Schultz.

© 2018 John Wiley & Sons Ltd. Published 2018 by John Wiley & Sons Ltd.



**Figure 10.1** Relationships Window for plumbing supply store database.

In the simplest case, *expressions* is a single field name and *data\_source* is a single table. For example, for the plumbing store database, the statement

```
SELECT OrderDate FROM ORDERS;
```

shows the *OrderDate* field for each row of the ORDERS table. The *expressions* specifier may also be a list of attribute names separated by commas. An example of such a query from the same database is:

```
SELECT City, State FROM CUSTOMER;
```

This query shows the *City* and *State* fields for each row of the CUSTOMER table. By default, SELECT does not eliminate duplicate rows in its output: for example, if there are three customers in the same city, then that city would appear three times in the output. However, SELECT has an optional DISTINCT modifier that removes any duplicate rows from the output table. Suppose we modify the above query to:

```
SELECT DISTINCT City, State FROM CUSTOMER;
```

The output of this query will consist of one row for each city in which customers are located, even if there is more than one customer in the same city.

A special possible value of *expressions* is “\*”, which displays every attribute in *data\_source*. For example:

```
SELECT * FROM CUSTOMER;
```

displays the entire CUSTOMER table. The items in the *expressions* specifier need not be simple attribute names but may also be computed expressions. For example, the query

```
SELECT ProductName, UnitsInStock*UnitPrice FROM PRODUCT;
```

displays the name of each product in the `PRODUCT` table, along with the total value of inventory of the product on hand, computed as the product of the number of units in stock and the unit price. The syntax of the expressions allowed in the `SELECT` clause is similar to that of Microsoft Excel formulas, with “\*” standing for multiplication. In the Access output for this query, the first column has the understandable heading “Product Name,” but the second one has the cryptic heading “Expr1001.” To provide a more understandable heading, we can add an `AS` modifier to the *expressions* specifier, as follows:

```
SELECT ProductName,  
       UnitsInStock*UnitPrice AS InventoryValue  
FROM   PRODUCT;
```

This modification causes the second column to have the more understandable heading “InventoryValue.” Note that there are no quotes or other special formatting around `InventoryValue` in this statement: the `AS` modifier effectively defines a new field name, which is formatted like any other field name. Here, we have broken the query into multiple lines to fit it on the page. You can insert line breaks and spaces anywhere in an SQL statement without changing its meaning, except in the middle of a word, table/field identifier, or number.

## WHERE Conditions

Often you do not want to display data from every row of a table but only from rows that meet certain criteria. That is the purpose of SQL’s `WHERE` clause. The most basic form of a `SELECT` query with a `WHERE` clause is:

```
SELECT expressions FROM data_source  
WHERE logical_expression ;
```

This statement functions much like the simplest `SELECT` statement, except that only information from the rows of *data\_source* for which *logical\_expression* evaluates to “true” appear in the output. A simple example of such a statement is:

```
SELECT ProductName FROM PRODUCT  
WHERE UnitsInStock >= 100;
```

This query displays the name of each product of which we have at least 100 units in stock. The *logical\_expression* following WHERE may be arbitrarily complicated. For example:

```
SELECT ProductName
FROM   PRODUCT
WHERE  UnitsOnOrder*UnitPrice > 5000;
```

shows the name of each product for which the total value of inventory on order is over \$5,000. In addition to simple comparisons using = (equal), > (greater than), < (less than), <= (less than or equal to), and >= (greater than or equal to), SQL allows compound logical expressions constructed through AND and OR operations. For example:

```
SELECT ProductName, UnitsOnOrder, UnitPrice
FROM   PRODUCT
WHERE  UnitsOnOrder >= 100 OR UnitPrice < 50;
```

displays the name, number of units on order, and unit price of each product that has at least 100 units on order or has a price less than \$50. Note that OR in SQL is “inclusive,” as in most computer languages, so that a record meeting both sub-conditions is considered to satisfy the OR condition. In the query above, for example, a product with at least 100 units on order and also costing less than \$50 would be displayed in the query output. Here is another example of a compound condition:

```
SELECT FirstName, LastName
FROM   CUSTOMER
WHERE  City="Hamilton" AND State="NJ";
```

This query will display the names of all customers from Hamilton, NJ; the database also contains a customer from Hamilton, NY, but this customer is not included in the query result because the *State* field value does not match the condition. When comparing text fields to literal character strings such as “CA,” you should enclose the literal character strings in double quotes. Otherwise, SQL will try to interpret the character string as an attribute name.

## Inner Joins

So far, this chapter has considered only queries drawn from a single table. We now discuss how SQL can express queries based on data from multiple tables. The most common technique for basing queries on multiple tables is called an

*inner join*. An inner join consists of all combinations of rows selected from two tables that meet some matching condition, formally called a *join predicate*. One standard syntax for an inner join, of which we have already seen examples earlier in this book, is:

```
First_Table INNER JOIN Second_Table ON Condition
```

Formally, this syntax specifies that the query should form a table consisting of all combinations of a record from *First\_Table* with a record from *Second\_Table* for which *Condition* evaluates to “true.” Most frequently, *Condition* specifies that a foreign key in one table should match a primary key in the other.

Here is an example of an inner join based on the plumbing store database:

```
SELECT FirstName, LastName, OrderDate
FROM   CUSTOMER INNER JOIN ORDERS
        ON CUSTOMER.CustomerID = ORDERS.CustomerID;
```

The INNER JOIN expression is now the *data\_source* following the FROM keyword, where before we used a single table. This construction means that the data to be displayed is taken from the temporary table resulting from the inner join operation. The particular inner join expression, namely,

```
CUSTOMER INNER JOIN ORDERS
    ON CUSTOMER.CustomerID = ORDERS.CustomerID
```

specifies that the query should be based on all combinations of records from the CUSTOMER and ORDERS tables that have matching *CustomerID* fields. This kind of primary key to foreign key matching condition is by far the most common kind of inner join. CUSTOMER.CustomerID refers to the *CustomerID* field from the CUSTOMER table, while ORDERS.CustomerID refers to the *CustomerID* field from the ORDERS table. The use of “.” here is called *qualification*. It is required to eliminate ambiguity whenever several underlying tables have fields of the same name, as is the case for the *CustomerID* in this example: if we were to just write CustomerID, SQL would not be able to tell whether we were referring to the *CustomerID* field in the CUSTOMER table or the *CustomerID* field in the ORDERS table. To resolve this ambiguity, we preface an attribute name with a table name and “.”: for example, CUSTOMER.CustomerID means the *CustomerID* attribute of the CUSTOMER table.

Qualification is not required for field names that occur in only one of the underlying tables. While it can still be used in such cases – for example, one could say CUSTOMER.FirstName instead of FirstName in this query – it is

not necessary, because there can be no ambiguity about which *FirstName* field is meant since this name occurs in only one underlying table.

The effect of this query is to display the date of each order, preceded by the first and last names of the corresponding customer (Table 10.1).

Just as for queries derived from just one table, we can use a WHERE clause to narrow the results of join-based queries. For instance, if we want to see the same information, but only for orders placed on or after April 28, 2013, we could write:

**Table 10.1** Output from first SQL inner join example query.

First Name	Last Name	Order Date
Benjamin	Masterson	4/20/2013
Benjamin	Masterson	4/21/2013
Benjamin	Masterson	4/22/2013
Benjamin	Masterson	4/24/2013
Mary	Milgrom	4/21/2013
Mary	Milgrom	4/22/2013
Mary	Milgrom	4/22/2013
Leonard	Goodman	4/18/2013
Margerita	Colon	4/15/2013
Margerita	Colon	4/24/2013
Geoffrey	Hammer	4/18/2013
Geoffrey	Hammer	4/25/2013
Geoffrey	Hammer	4/26/2013
Geoffrey	Hammer	5/1/2013
Geoffrey	Hammer	5/1/2013
Ashley	Flannery	4/18/2013
Ashley	Flannery	4/24/2013
Joseph	Brower	4/30/2013
Xiaoming	Wang	4/25/2013
Derek	Escher	4/29/2013
Derek	Escher	4/29/2013
Laura	Ng	4/26/2013
Laura	Ng	4/26/2013
Robert	Sloan	4/27/2013
Robert	Sloan	4/28/2013

```
SELECT FirstName, LastName, OrderDate
FROM    CUSTOMER INNER JOIN ORDERS
        ON CUSTOMER.CustomerID = ORDERS.CustomerID
WHERE   OrderDate >= #4/28/2013#;
```

This query demonstrates the special syntax that SQL uses for dates, which is also used in the Access query grid: we enclose any date between “#” characters. Without this special syntax, SQL would mistake “4/28/2013” for the number 4 divided by 28, and then divided again by 2013.

When queries are based on more than one table, the “\*” syntax can still be used to indicate “all fields.” For example, we may write:

```
SELECT *
FROM    CUSTOMER INNER JOIN ORDERS
        ON CUSTOMER.CustomerID = ORDERS.CustomerID
WHERE   OrderDate >= #4/28/2013#;
```

In response, SQL displays all rows of the inner-joined table whose order date is on or after April 28, 2013. There is also a qualified form of “\*”. For instance, the query

```
SELECT FirstName, LastName, ORDERS.*
FROM    CUSTOMER INNER JOIN ORDERS
        ON CUSTOMER.CustomerID = ORDERS.CustomerID
WHERE   OrderDate >= #4/28/2013#;
```

shows the customer first name, customer last name, and all fields from the ORDERS table for orders placed on or after April 28, 2013. Here, `ORDERS.*` means “all fields from the ORDERS table.”

## Cartesian Joins and a Different Way to Express Inner Joins

In addition to INNER JOIN expressions, the FROM clause of a SELECT statement may contain a list of table names separated by commas. For example, it is possible to write:

```
SELECT FirstName, LastName, OrderDate
FROM    CUSTOMER, ORDERS;
```

However, this query does *not* behave the same way as the first query in the previous section. The construction “CUSTOMER, ORDERS” in the FROM clause instructs SQL to form the table obtained by combining every possible

row of the CUSTOMER table with every possible row of the ORDERS table, without checking whether the foreign key values linking the two tables match. Thus, if the CUSTOMER table contains 14 rows and the ORDERS table contains 25 rows, the resulting table contains  $14 \times 25 = 350$  rows, most of which are essentially meaningless. This kind of join is called a *Cartesian join* or *cross join*.

The Cartesian join may seem like an unnatural way to interpret the above query, but the rationale is that an SQL statement's meaning should be the same no matter what relationships the database designer may have intended to exist between tables. More precisely, some early relational database systems maintained metadata describing the structure of each table but did not necessarily store metadata explicitly describing foreign keys. The interpretation of a query like the one above therefore needed to be independent of any intended foreign key relationships.

Cartesian joins are rarely useful in isolation but can be useful building blocks in various data manipulations. Typically one uses a WHERE clause to narrow down the result of a Cartesian join to something more useful. For example, we may change the above query to:

```
SELECT FirstName, LastName, OrderDate
FROM   CUSTOMER, ORDERS
WHERE  CUSTOMER.CustomerID = ORDERS.CustomerID;
```

This query specifies that SQL form the Cartesian join of the CUSTOMER and ORDER tables but then remove all rows violating the condition

```
CUSTOMER.CustomerID = ORDERS.CustomerID
```

The result is exactly the same temporary table as the INNER JOIN operation in the query in the previous section. The SELECT clause then specifies that SQL display first name, last name, and order date attributes from this temporary table, so the query generates exactly the same result as the first one in the previous section.

If interpreted literally, one might expect this query to run more slowly and require more memory than its equivalent INNER JOIN form, because of the potentially gigantic size of the Cartesian join table specified by the expression CUSTOMER, ORDERS. However, most SQL interpreters have a query optimizer module that analyzes queries and attempts to find efficient ways of producing the specified results. Most query optimizers would recognize that this query is equivalent to an inner join and execute it at approximately the same speed. In fact, when SQL was originally created, it had only WHERE; the explicit INNER JOIN syntax was introduced later.

To further illustrate the concept of a Cartesian join, temporarily consider the following very simple database:


```
PERSON(PersonID, FirstName, LastName)
EVENT(EventID, EventTime, PersonID)
      PersonID foreign key to PERSON
```



Suppose we execute the Cartesian join query:

```
SELECT * FROM PERSON, EVENT;
```

This query will create a temporary table from all possible combinations of a PERSON record and EVENT records, regardless of whether the foreign key *PersonID* has a matching value. The result of the query is illustrated in Figure 10.2 for the case that the PERSON table has two records and the EVENT table has three records.



PERSON		
PersonID	FirstName	LastName
P0001	Katya	Elmberg
P0002	Andrew	Kim

EVENT		
EventID	EventTime	PersonID
E0001	4/1/2016 4:32 PM	P0002
E0002	3/6/2016 9:17 AM	P0001
E0003	4/2/2016 3:22 PM	P0002

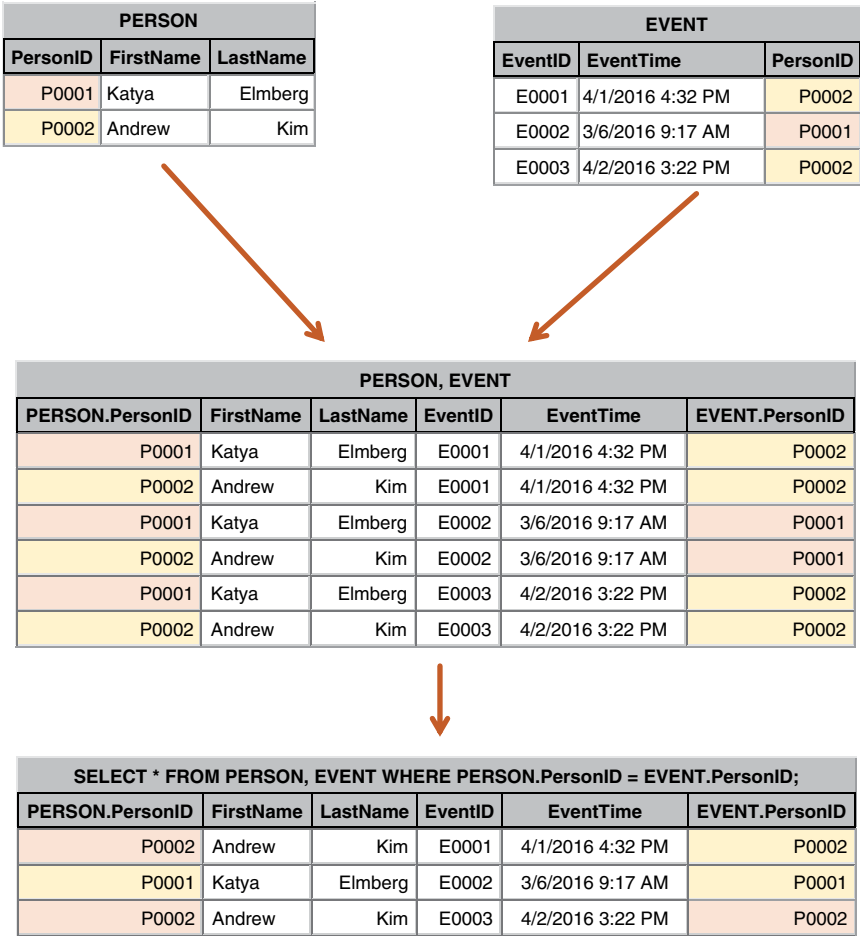
PERSON, EVENT					
PERSON.PersonID	FirstName	LastName	EventID	EventTime	EVENT.PersonID
P0001	Katya	Elmberg	E0001	4/1/2016 4:32 PM	P0002
P0002	Andrew	Kim	E0001	4/1/2016 4:32 PM	P0002
P0001	Katya	Elmberg	E0002	3/6/2016 9:17 AM	P0001
P0002	Andrew	Kim	E0002	3/6/2016 9:17 AM	P0001
P0001	Katya	Elmberg	E0003	4/2/2016 3:22 PM	P0002
P0002	Andrew	Kim	E0003	4/2/2016 3:22 PM	P0002

**Figure 10.2** Example of a Cartesian join.

The query result has  $6 = 2 \times 3$  rows, with half of them having non-matching *PersonID* fields. Suppose that we instead execute the query:

```
SELECT *
FROM   PERSON, EVENT
WHERE  PERSON.PersonID = EVENT.PersonID;
```

This query takes the result of the Cartesian join query above and removes the rows with non-matching *PersonID* fields. This process is depicted in Figure 10.3.



**Figure 10.3** Example of a Cartesian join being reduced to an inner join through a WHERE condition.

The final result is the most common way to join the PERSON and EVENT tables, combining only pairs or rows that have common *PersonID* values. The result simply looks like the EVENT table, with the corresponding information from the PERSON table prepended to each row. It is identical to the output of the query:

```
SELECT *
FROM    PERSON INNER JOIN EVENT
        ON PERSON.PersonID = EVENT.PersonID;
```

The main advantage of expressing inner joins through a Cartesian join and WHERE clauses occurs when queries are based on more than two tables.

Suppose, returning to the plumbing store database example, that we want to display the *Quantity* field from each row of the ORDERDETAIL table, along with the name of the corresponding product and the date of the order. We may write this query as:

```
SELECT Quantity, ProductName, OrderDate
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  ORDERS.OrderID = ORDERDETAIL.OrderID AND
       ORDERDETAIL.ProductID = PRODUCT.ProductID;
```

Literally, the FROM clause in this query specifies that SQL form all possible combinations of rows from the three tables ORDERS, ORDERDETAIL, and PRODUCT, but the WHERE clause next indicates that it should only retain those combinations for which both foreign keys match. If interpreted literally, this query might be very slow and consume a huge amount of space. Properly analyzed by a query optimizer, however, it should run at about the same speed as the equivalent query constructed from inner joins, which looks like this:

```
SELECT Quantity, ProductName, OrderDate
FROM   (ORDERS INNER JOIN ORDERDETAIL
        ON ORDERS.OrderID = ORDERDETAIL.OrderID)
       INNER JOIN PRODUCT
        ON ORDERDETAIL.ProductID = PRODUCT.ProductID;
```

The first INNER JOIN clause in this query creates a temporary table that combines records from the ORDERS and ORDERDETAIL tables based on matching the *OrderID* foreign keys. The second INNER JOIN clause then combines rows of the resulting temporary table with rows of the PRODUCT table based on matching values of the *ProductID* field. The result is identical to the Cartesian join and WHERE combination in the previous example, but the SQL code may be harder for a human to read. Furthermore, if one omits the parentheses surrounding the first INNER JOIN clause, Access SQL will generate an error message instead of processing the query. If you inspect the SQL code automatically written by Access to implement queries constructed using the design grid, you will often see multiple inner join constructions like the one above, with multiple levels of nested parentheses if there are more than three tables. In this and the next chapter, we will concentrate on writing SQL directly, and will therefore prefer the more human-understandable form using WHERE. Some other dialects of SQL do not require parentheses when combining multiple INNER JOIN clauses, making the INNER JOIN option more attractive. Some IT departments therefore prefer using INNER JOINS because their efficiency is less reliant on query optimizer modules.

A compound WHERE clause can do “double duty” by both performing a join and filtering records. For example, suppose we want to perform the same query

as above but wanted to see only cases in which at least 10 units of a product were in the same order. We could then write:

```
SELECT Quantity, ProductName, OrderDate
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID AND
        Quantity >= 10;
```

This query produces the results shown in Table 10.2.

**Table 10.2** Results of query selecting products appearing at least 10 times in the same order.

Quantity	Product Name	Order Date
10	Replacement Valve Units Type A	4/21/2013
10	Frost-Free Outdoor Faucet Set	4/24/2013
10	Retro Nickel Bath/Shower Combo	4/24/2013
10	Flexible Spray Shower	4/24/2013
10	Budget Bath Sink Set	4/24/2013
25	Replacement Valve Units Type A	4/27/2013

Using WHERE to implement inner joins does have a potential pitfall, the unintentional Cartesian join. Suppose we want to use a more complicated, compound filtering condition in the above query: we want to see all order lines where the quantity is at least 10 or the order was placed on May 1, 2013. It may seem natural to write:

```
SELECT Quantity, ProductName, OrderDate
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID AND
        Quantity >= 10 OR OrderDate = #5/1/2013#;
```

There are only six records in the ORDERDETAIL table corresponding to the two orders placed on May 1, 2013, so one might expect this query to produce at most six more rows than the previous one. But instead it produces thousands of rows. The reason is that SQL, like most computer languages, processes AND before OR when interpreting compound logical expressions.<sup>1</sup>

Despite the way that we placed spaces and line breaks in the above query (which cannot alter its meaning), SQL interprets the above query as:

<sup>1</sup> The reason for this grouping is that AND and OR have analogous mathematical properties to multiplication and addition: essentially, AND is the multiplication of logic, while OR is the addition of logic.

```

SELECT Quantity, ProductName, OrderDate
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  (ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID AND
        Quantity >= 10)
        OR OrderDate = #5/1/2013#;

```

Thus, SQL forms the Cartesian join of the three tables `ORDERS`, `ORDERDETAIL`, and `PRODUCT`, which has thousands of rows, and then places two kinds of rows from this Cartesian join into the output:

- Rows where all the foreign keys match and the *Quantity* field is at least 10.
- Rows for which the order date is May 1, 2013, without any check that the foreign keys match.

The second variety of rows fills the query output with useless or misleading information. The solution to this misbehavior is simple: place parentheses within the `WHERE` condition to make sure it processes in the proper order. Therefore, we write:

```

SELECT Quantity, ProductName, OrderDate
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID AND
        (Quantity >= 10 OR OrderDate = #5/1/2013#);

```

This small change restores sanity to the query output, producing the results in Table 10.3.

**Table 10.3** Results of OR query after correcting inadvertent Cartesian join.

Quantity	Product Name	Order Date
10	Replacement Valve Units Type A	4/21/2013
10	Frost-Free Outdoor Faucet Set	4/24/2013
10	Retro Nickel Bath/Shower Combo	4/24/2013
10	Flexible Spray Shower	4/24/2013
10	Budget Bath Sink Set	4/24/2013
25	Replacement Valve Units Type A	4/27/2013
1	Omnidirectional Shower	5/1/2013
5.00	Massage Shower System	5/1/2013
3.00	Replacement Valve Units Type A	5/1/2013
1.00	Budget Bath Sink Set	5/1/2013
2.00	Spacesaver Toilet	5/1/2013
2.00	Retro Nickel Bath/Shower Combo	5/1/2013

## Aggregation

In addition to the features we have discussed so far, SQL also allows queries to perform *aggregation*, which means computing summary information about all records or groups of records. Aggregation occurs when you use any aggregation function in your query. The most commonly used aggregation functions in SQL are shown in Table 10.4.

**Table 10.4** Common SQL aggregation functions.

Function	Meaning
Sum( )	Sum
Avg( )	Average (sample mean)
Count( )	Number of non-blank data items
Min( )	Smallest or alphabetically first value
Max( )	Largest or alphabetically last value
StDev( )	Sample standard deviation
First( )	First value encountered <sup>2</sup>
Last( )	Last value encountered <sup>2</sup>

To give a simple example of using such functions, suppose we want to know the total number of items ordered over the entire history covered by the database. To display this information, we may write the query:

```
SELECT Sum(Quantity) FROM ORDERDETAIL;
```

This query adds up the *Quantity* field over all rows of the ORDERDETAIL table and produces (for the particular data in the example from Chapter 7) the single result “163.” It is also possible to specify multiple aggregation operations in the same query. By way of illustration:

```
SELECT Sum(Quantity), Avg(Quantity), Max(OrderID)
FROM ORDERDETAIL;
```

displays a single row of information showing the total number of items ordered, the average number of items per order detail line, and the alphabetically last *OrderID* value for which there are any order detail records (perhaps not a particularly useful piece of information).

<sup>2</sup> The results of `First()` and `Last()` may be somewhat arbitrary, because the basic principles of relational databases do not allow full control over the order in which rows of a table are processed.

We may also apply aggregation functions to expressions rather than to just simple fields. For example, if we want to know the average difference (over all products) between the units in stock and the units on order, we can use the query:

```
SELECT Avg(UnitsinStock - UnitsonOrder) FROM PRODUCT;
```

Here, we are using the Avg aggregation function instead of the Sum aggregation function and a more complicated expression than a simple field as its argument. This query produces the single value 29.6451612903, meaning that the number of units in stock averages about 30 units higher than the number of units on order.<sup>3</sup>

Now suppose that we want to see the average order quantity per order line for products whose *ProductID* values are between 1 and 13. To accomplish this, we write:

```
SELECT Avg(Quantity)
FROM   ORDERDETAIL
WHERE  ProductID >= 1 AND ProductID <= 13;
```

WHERE clauses always filter data source rows *before* any aggregation is performed, so that the average is now taken only for those products between P0001 and P0013 (inclusive), rather than all possible product IDs.<sup>4</sup> Incidentally, since applying a lower and upper limit to the same attribute is very common, SQL provides a special logical operator BETWEEN, which takes the form “*x* BETWEEN *a* AND *b*” and is equivalent to “*x* >= *a* AND *x* <= *b*”. You can use this kind of expression anywhere one might use a logical condition. Thus, we can also write the query as:

```
SELECT Avg(Quantity)
FROM   ORDERDETAIL
WHERE  ProductID BETWEEN 1 AND 13;
```

This form of the query produces exactly the same result as the previous one (1.77272727273).

---

3 Since an average of a difference is identical to a difference of averages, we could also have expressed this query as `SELECT Avg(UnitsinStock) - Avg(UnitsonOrder) FROM PRODUCT;`

4 Since the *ProductID* field has the Long Integer datatype and the “P” symbols here are only part of the format specifier and not stored in the *ProductID* field, SQL manipulates *ProductID* values like numbers rather than character strings. Therefore, we do not use quotes and do not include a “P” when expressing SQL comparisons for *ProductID* values.

Another useful aggregation function is `Count`, which simply counts the number of non-blank data items in its argument. For example, the query

```
SELECT Count(OrderID)
FROM   ORDERS
WHERE  OrderDate BETWEEN #4/22/2013# AND #4/28/2013#;
```

counts the number of orders received between April 22 and 28, 2013 (inclusive). The `Count` function should not be confused with the `Sum` function. Within a given set of records, `Count` will produce exactly the same result when applied to any argument that cannot be blank: in the above situation, it simply counts the number of records meeting the order date criterion. For example, the two queries

```
SELECT Count(ProductID) FROM PRODUCT;
SELECT Count(UnitsOnOrder) FROM PRODUCT;
```

will produce exactly the same result, namely the total number of rows in the `PRODUCT` table, as long as we do not allow the *UnitsOnOrder* field to contain blanks. A zero value is not considered to be blank, so even products with zero units on order will still contribute to the count. To add up the total number of units on order across all products, we would instead use the query:

```
SELECT Sum(UnitsOnOrder) FROM PRODUCT;
```

This query produces a totally different result from that using `Count`. In the next chapter, we will show some more advanced examples that take advantage of the fact that `Count` does not include blanks.

We may also apply aggregation across the rows of joined tables. Suppose we want to know the total revenue from orders placed between April 22 and April 29, 2013 (inclusive). We may calculate this information with the following query:

```
SELECT Sum(UnitPrice*Quantity)
FROM   ORDERS, ORDERDETAIL, PRODUCT
WHERE  ORDERS.OrderID = ORDERDETAIL.OrderID AND
       ORDERDETAIL.ProductID = PRODUCT.ProductID AND
       OrderDate BETWEEN #4/22/2013# AND #4/29/2013#;
```

Here, the query applies aggregation to the result of a join operation. We start the query by (in principle) forming a Cartesian join with the `ORDERS`, `ORDERDETAIL`, and `PRODUCT` tables, and then use the same standard join conditions as in the previous section:

```
ORDERS.OrderID = ORDERDETAIL.OrderID AND
ORDERDETAIL.ProductID = PRODUCT.ProductID
```



These conditions ensure that we match up only triples of records that “make sense” together. The result consists basically of each row of the ORDERDETAIL table, augmented by the related information from the ORDER and PRODUCT tables. We then filter out the elements outside the date range that interests us with the additional condition:

```
AND OrderDate BETWEEN #4/22/2013# AND #4/29/2013#
```

This additional stipulation once again illustrates how both join-related conditions and other forms of criterion selection can be mixed in the same compound WHERE clause. It also once again shows how SQL uses “#” symbols around dates. The line

```
SELECT Sum(UnitPrice*Quantity)
```

indicates that SQL should take the sum of *UnitPrice* times *Quantity* over all the records of the joined and filtered table, resulting in a total of \$36,583.65. This query is another example of applying aggregation to a computed expression rather than a simple field.

## GROUP BY

More often than not, we do not want to aggregate information over the entire dataset but over multiple groups of records. Suppose that in the previous query, we do not want a single grand total but the total revenue for each day. To perform this kind of processing, we use another standard SQL feature, the GROUP BY clause. Suppose that we make the following additions to the previous query:

```
SELECT  OrderDate, Sum(UnitPrice*Quantity)
FROM    ORDERS, ORDERDETAIL, PRODUCT
WHERE   ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID AND
        OrderDate BETWEEN #4/22/2013# AND #4/29/2013#
GROUP BY OrderDate;
```

The most important addition here is the new clause “GROUP BY OrderDate.” It works as follows: once the tables have been joined and the WHERE conditions have been applied, the query divides the rows of the joined and filtered table into groups. In this case, the GROUP BY clause specifies only the single expression *OrderDate*, so records are in the same group when they have the same order date. Each group is then condensed into a single row of query output, so we get as many rows of output as there are groups.

When GROUP BY is specified, aggregate functions like Sum no longer operate across the entire dataset but separately on each group. Thus, the query adds up *UnitPrice* times *Quantity* for all the records on a given date and reports a separate sum for each date, as shown in Table 10.5.

**Table 10.5** Output of query computing revenue on each date.

Order Date	Expr1001
4/22/2013	\$24,072.95
4/24/2013	\$7,833.70
4/25/2013	\$359.75
4/26/2013	\$1,082.60
4/27/2013	\$249.75
4/28/2013	\$569.95
4/29/2013	\$2,414.95

The “Expr1001” heading on the second columns is not very explanatory, a minor issue we can fix by adding an optional AS clause:

```
SELECT    OrderDate, Sum(UnitPrice*Quantity) AS Revenue
FROM      ORDERS, ORDERDETAIL, PRODUCT
WHERE     ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID AND
          OrderDate BETWEEN #4/22/2013# AND #4/29/2013#
GROUP BY OrderDate;
```

This query produces the more aesthetic output shown in Table 10.6.

**Table 10.6** Query computing revenue on each date, using AS to obtain a more understandable column name.

Order Date	Revenue
4/22/2013	\$24,072.95
4/24/2013	\$7,833.70
4/25/2013	\$359.75
4/26/2013	\$1,082.60
4/27/2013	\$249.75
4/28/2013	\$569.95
4/29/2013	\$2,414.95

We include the *OrderDate* field in the `SELECT` clause so that each row of output includes the date to which it applies. Otherwise, we would just see a column of revenue totals without any dates attached.

Conceptually, this query works by first forming a huge Cartesian join on the `ORDERS`, `ORDERDETAIL`, and `PRODUCT` tables, discarding all the resulting rows that have non-matching foreign keys or do not meet the date window condition, then forming groups, and finally summing up *UnitPrice* times *Quantity* over each group. In practice, and depending on the exact database software that you use, a query optimizer module in the SQL language processing system might rearrange the query's operations to make it more efficient. For example, it might filter the `ORDERS` table based on order date *before* performing the join.

Next, suppose we want to display the total revenue for each day recorded in the database, but organized by customer rather than by date, and showing the customer's first and last name. We may attempt to implement this query as follows:

```
SELECT  FirstName, LastName,
        Sum(UnitPrice*Quantity) AS Revenue
FROM    CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE   CUSTOMER.CustomerID = ORDERS.CustomerID AND
        ORDERS.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY FirstName, LastName;
```

We now join records from all four tables in the database, based on all the respective foreign keys matching – we need to include the `CUSTOMER` table in this query, because we need to output the name of the customer, which resides only in that table. More importantly, this query illustrates the use of `GROUP BY` with more than one attribute. When multiple attributes are specified in a `GROUP BY` clause, groups are formed based on *all* the specified attributes being identical. In this case, two records of the joined table are placed into the same group if both their *FirstName* and *LastName* fields have identical contents. If any of their grouped-by fields are not the same, the query places the records into different groups. For example, the above query would place data for two people with the same last name but different first names into different groups.

For a larger store than the one in our database, the above query has a high chance of producing misleading results because two customers might well have the same name. For example, if our database contained two customers named “William Jones,” their purchases would form a single group, and they would appear to be a single customer in the query output. To prevent this behavior, we modify the query as follows:

```
SELECT    FirstName, LastName,
          Sum(UnitPrice*Quantity) AS Revenue
FROM      CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE     CUSTOMER.CustomerID = ORDERS.CustomerID AND
          ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY  CUSTOMER.CustomerID, FirstName, LastName;
```

This query specifies that two records be placed in the same group only if they have the same *CustomerID*, the same first name, and the same last name. Since no two customers can have the same *CustomerID*, two different customers can no longer be placed in the same group. If there were two customers with the same name, they would appear on different lines of the output. We obtain the output shown in Table 10.7.

**Table 10.7** Output of query showing revenue aggregated by customer.

First Name	Last Name	Revenue
Benjamin	Masterson	\$7,293.60
Mary	Milgrom	\$23,783.75
Leonard	Goodman	\$3,982.95
Margerita	Colon	\$5,141.26
Geoffrey	Hammer	\$8,670.27
Ashley	Flannery	\$8,246.50
Joseph	Brower	\$225.85
Xiaoming	Wang	\$249.90
Derek	Escher	\$2,414.95
Laura	Ng	\$822.70
Robert	Sloan	\$819.70

Since the query contains the WHERE constraint `CUSTOMER.CustomerID = ORDERS.CustomerID`, it does not matter if we group by `CUSTOMER.CustomerID` or `ORDERS.CustomerID` because, after applying the WHERE criteria, these attributes will have the same value in every retained row of the joined table. However, we must still apply some qualification to this identifier, because there are two tables mentioned in the query that have a field called *CustomerID*: if we just tried to group on *CustomerID* with no qualification, we would receive an error message (at least in the Access dialect of SQL).

It may seem redundant that we are grouping by the combination of *CustomerID*, *FirstName*, and *LastName*, because *CustomerID* determines

*FirstName* and *LastName*. That is, if two records have the same *CustomerID* value, then they must have the same *FirstName* and *LastName* values. Thus, while the query says “GROUP BY CUSTOMER.CustomerID, FirstName, LastName”, it is effectively grouping only by *CustomerID*. In Access, however, if we try to simplify the query to

```
SELECT    FirstName, LastName,
          Sum(UnitPrice*Quantity) AS Revenue
FROM      CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE     CUSTOMER.CustomerID = ORDERS.CustomerID AND
          ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY  CUSTOMER.CustomerID;
```

we get a rather cryptic error message relating to the attribute *FirstName*. The problem is that standard SQL must be able to verify a query’s correctness without knowing the primary and foreign key structure of the underlying database. Therefore, Access’ SQL language “parser” does not take into account that *CustomerID* determines *FirstName* and *LastName*, and thus cannot verify whether the query above has an unambiguous meaning. As a result, it produces an error message instead of processing the query. In general, there is a rule that whenever any form of aggregation is present, every attribute appearing in the SELECT part of the query must be either within an aggregation function (like Sum), or be grouped by.

While this restriction is part of standard SQL, some database systems do not enforce it. For example, the above query would run and produce the expected results in the popular open-source MySQL system that is built into many websites. When it encounters a non-aggregated and non-grouped-by attribute in the SELECT clause, MySQL will just select an arbitrary representative of the group to display. In cases like the one above, where every member of any given group must have exactly the same value of *FirstName* and *LastName*, the only value that could possibly be displayed is the one that we would expect.

However, if we display just the *CustomerID* rather than the first and last name, then there would be no problem in any version of SQL. That is, the query

```
SELECT    CUSTOMER.CustomerID,
          Sum(UnitPrice*Quantity) AS Revenue
FROM      CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE     CUSTOMER.CustomerID = ORDERS.CustomerID AND
          ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY  CUSTOMER.CustomerID;
```

runs with no problem in Access, although the output is not as human-friendly as before, showing only customer IDs but no name information (see Table 10.8).

**Table 10.8** Output of simpler query computing revenue per customer, but identifying customers only by *CustomerID*.

CustomerID	Revenue
C0001	\$7,293.60
C0002	\$23,783.75
C0003	\$3,982.95
C0004	\$5,141.26
C0006	\$8,670.27
C0007	\$8,246.50
C0008	\$225.85
C0009	\$249.90
C0010	\$2,414.95
C0011	\$822.70
C0012	\$819.70

As an aside, this version of the query is unnecessarily complicated, because it does not really need the CUSTOMER table, since the *CustomerID* attribute is already present in ORDERS. Dispensing with the CUSTOMER table, we obtain exactly the same output through the simpler query:

```
SELECT    CustomerID, Sum(UnitPrice*Quantity) AS Revenue
FROM      ORDERS, ORDERDETAIL, PRODUCT
WHERE     ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY CustomerID;
```

But if we wish to display customer name information, we must use the CUSTOMER table. Returning to the version of the query that displays customer names, an alternative to using a seemingly redundant GROUP BY clause is to apply some unnecessary aggregation operation in the SELECT clause. For example, we could write:

```
SELECT    Min(FirstName), Min(LastName),
          Sum(UnitPrice*Quantity) AS Revenue
FROM      CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE     CUSTOMER.CustomerID = ORDERS.CustomerID AND
          ORDERS.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY CUSTOMER.CustomerID;
```

The `Min` operation, when applied to text data, selects the alphabetically first value within each group. But since *FirstName* and *LastName* take the same value throughout each group, the output for each group contains the only possible applicable first name and last name. This query is probably more confusing to a human than the original one, however. If we were to take this approach, we would also need to use more `AS` modifiers to make the column names in the output more readable.

We do not give an example here, but it is possible to `GROUP BY` not only by the values of simple attributes, but the values of general expressions (computed fields).

When you use `GROUP BY`, one should have at least one aggregation function such as `Sum` or `Avg` somewhere in your query, typically in the `SELECT` clause. Without any aggregation functions, `GROUP BY` will either cause an error message, have no effect, or operate in the same way as `SELECT DISTINCT`, which is simpler to use.

## HAVING

Suppose we are interested in performing the same query, but we want to see only customers who have spent a total of at least \$7,000. This restriction cannot be imposed by a `WHERE` clause, because `WHERE` restrictions are always applied *before* grouping and aggregation occur. Here, we need to instead apply a criterion to the *result* of the `Sum` aggregation function, which can only be known after the grouping and aggregation steps. To apply criteria after grouping and aggregation, SQL provides an additional clause called `HAVING`. To implement this query, we add the clause

```
HAVING Sum(UnitPrice*Quantity) >= 7000
```

to the query, resulting in:

```
SELECT  FirstName, LastName,
        Sum(UnitPrice*Quantity) AS Revenue
FROM    CUSTOMER, ORDERS, ORDERDETAIL, PRODUCT
WHERE   CUSTOMER.CustomerID = Orders.CustomerID AND
        Orders.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY CUSTOMER.CustomerID, FirstName, LastName
HAVING  Sum(UnitPrice*Quantity) >= 7000;
```

This clause specifies that, after grouping and aggregation, we retain only output rows for which the sum of the unit price multiplied by quantity is at least 7,000. Now, only “big spenders” appear in the output, as shown in Table 10.9.

**Table 10.9** Output of query identifying high-spending customers.

First Name	Last Name	Revenue
Benjamin	Masterson	\$7,293.60
Mary	Milgrom	\$23,783.75
Geoffrey	Hammer	\$8,670.27
Ashley	Flannery	\$8,246.50

HAVING and WHERE perform similar functions, but WHERE filters records *before* aggregation and HAVING filters records *after* aggregation. Both can be used in the same aggregation query (as is the case above). In queries without aggregation, there is no difference between HAVING and WHERE, but it is customary to use only WHERE. The logical expression appearing after HAVING should customarily include an aggregation function, since otherwise we could accomplish the same result with WHERE.

Since the computed field *Revenue* is already defined by an AS clause, one might expect to be able to substitute “HAVING Revenue >= 7000” for the HAVING clause in the above query. Unfortunately, this technique does not work properly in Access, although it may work in other SQL dialects. In Access, AS modifiers within the SELECT clause do not influence expressions in other clauses, so we instead get a request to enter the value of the undefined parameter “Revenue.”

## ORDER BY

SQL’s ORDER BY clause sorts the output of a query and can be used in queries with or without aggregation. Despite its superficial resemblance to GROUP BY, it performs a completely different function: GROUP BY establishes groups over which aggregation functions operate, whereas ORDER BY places the final query results in a specified sequence. ORDER BY sorting happens after aggregation when aggregation is present, so it occurs after the grouping specified by GROUP BY. The existence of the ORDER BY clause is the reason our plumbing store database violates our usual naming conventions by calling order table ORDERS rather than ORDER. Because ORDER, being part of ORDER BY, is a “reserved word”<sup>5</sup> in SQL, calling a table ORDER can badly confuse the SQL parser.

For example, if we want to see “big spenders” in the previous query in order of spending, with biggest spenders first, we would write:

<sup>5</sup> In a computer language, a *reserved word* is one that has a predefined, special meaning and cannot be redefined by the programmer. For example, SELECT, AS, FROM, and WHERE are all reserved words in SQL. Using such words as the name of a table or attribute can badly confuse SQL and will result in syntax errors.



```

SELECT  FirstName, LastName,
        Sum(UnitPrice*Quantity) AS Revenue
FROM    CUSTOMER, Orders, ORDERDETAIL, PRODUCT
WHERE   CUSTOMER.CustomerID = Orders.CustomerID AND
        Orders.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY CUSTOMER.CustomerID, FirstName, LastName
HAVING  Sum(UnitPrice*Quantity) >= 7000
ORDER BY Sum(UnitPrice*Quantity) DESC;

```

This query's results are shown in Table 10.10.

**Table 10.10** Output of query showing high-spending customers sorted by expenditure.

First Name	Last Name	Revenue
Mary	Milgrom	\$23,783.75
Geoffrey	Hammer	\$8,670.27
Ashley	Flannery	\$8,246.50
Benjamin	Masterson	\$7,293.60

The DESC modifier here specifies a sort in descending order, meaning that the largest or alphabetically last values should appear first. Note that we can ORDER BY any combination of aggregated and non-aggregated expressions that could appear in the SELECT clause. For example, if we wanted to order the same query alphabetically primarily by last name, and then secondarily by first name,<sup>6</sup> we would change the query to:

```

SELECT  FirstName, LastName,
        Sum(UnitPrice*Quantity) AS Revenue
FROM    CUSTOMER, Orders, ORDERDETAIL, PRODUCT
WHERE   CUSTOMER.CustomerID = Orders.CustomerID AND
        Orders.OrderID = ORDERDETAIL.OrderID AND
        ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY CUSTOMER.CustomerID, FirstName, LastName
HAVING  Sum(UnitPrice*Quantity) >= 7000
ORDER BY LastName, FirstName;

```

This version of the query results in the different ordering shown in Table 10.11.

<sup>6</sup> That is, among people with the same last name, we order by first name.

**Table 10.11** High-spending customers sorted by their names.

First Name	Last Name	Revenue
Ashley	Flannery	\$8,246.50
Geoffrey	Hammer	\$8,670.27
Benjamin	Masterson	\$7,293.60
Mary	Milgrom	\$23,783.75

The sorting priority is always from left to right in the ORDER BY clause, but you can specify the expressions in the ORDER BY clause in a different sequence than in the SELECT clause. By default, numerical values are sorted from smallest to largest, dates are sorted from earlier to later, and text is sorted from alphabetically first to alphabetically last. The modifier DESC may be appended to any individual sort expression to reverse the default order. You may use the modifier ASC to specify the normal, ascending order, but only to clarify the intent of the query to human readers, since ascending order is the default.

You can use ORDER BY in queries either with or without aggregation. As an example of an ORDER BY query without aggregation, consider:

```
SELECT    FirstName, LastName, OrderDate
FROM      CUSTOMER, ORDERS
WHERE     CUSTOMER.CustomerID = ORDERS.CustomerID
ORDER BY  OrderDate DESC, LastName, FirstName;
```

This query displays the customer first name, customer last name, and date of each order, sorted from the most recent orders to the oldest. Orders placed on the same date are presented alphabetically by customer last name; orders placed on the same date by customers with the same last name are presented alphabetically by customer first name (although this consideration does not affect this particular small database). This query produces the output shown in Table 10.12.

## The Overall Conceptual Structure of Queries

In summary, SQL queries have the following general form:

```
SELECT {DISTINCT} expressions1  "{" means DISTINCT is
                                optional
FROM      data_source
WHERE     conditions1            Optional
GROUP BY expressions2            Optional
HAVING   conditions2            Optional
ORDER BY expressions3            Optional
;
```

**Table 10.12** Orders sorted by customer name and date.

First Name	Last Name	Order Date
Geoffrey	Hammer	5/1/2013
Geoffrey	Hammer	5/1/2013
Joseph	Brower	4/30/2013
Derek	Escher	4/29/2013
Derek	Escher	4/29/2013
Robert	Sloan	4/28/2013
Robert	Sloan	4/27/2013
Geoffrey	Hammer	4/26/2013
Laura	Ng	4/26/2013
Laura	Ng	4/26/2013
Geoffrey	Hammer	4/25/2013
Xiaoming	Wang	4/25/2013
Margherita	Colon	4/24/2013
Ashley	Flannery	4/24/2013
Benjamin	Masterson	4/24/2013
Benjamin	Masterson	4/22/2013
Mary	Milgrom	4/22/2013
Mary	Milgrom	4/22/2013
Benjamin	Masterson	4/21/2013
Mary	Milgrom	4/21/2013
Benjamin	Masterson	4/20/2013
Ashley	Flannery	4/18/2013
Leonard	Goodman	4/18/2013
Geoffrey	Hammer	4/18/2013
Margherita	Colon	4/15/2013

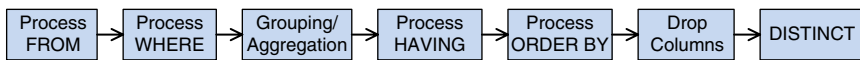
Conceptually, the sequence of operations is as follows:

- 1) We form the Cartesian join of the tables appearing in the FROM clause. This table consists of all possible combinations of records from the constituent tables of the query. If we use the INNER JOIN syntax instead of separating tables with commas, those specific inner joins are made.<sup>7</sup>

<sup>7</sup> In the next chapter, we will revisit the concept of an outer join, which SQL expresses using the LEFT JOIN or RIGHT JOIN syntax. Such joins also happen in the first step of the conceptual query sequence.

- 2) We drop any rows that do not meet the WHERE conditions. If the FROM clause is made up of Cartesian joins, the WHERE conditions typically include checks to make sure that foreign keys match, as well as any other desired conditions.
- 3) We perform any specified grouping and aggregation. We form groups based on the concatenated values of all the expressions in the GROUP BY clause, and then condense each group to a single row. Any expressions involving aggregation functions specified in the SELECT or HAVING clauses are computed at this time.
- 4) We apply the post-aggregation criteria specified by the HAVING clause, dropping (aggregated) rows that do not met the specified criteria.
- 5) We sort the resulting rows as specified in the ORDER BY clause.
- 6) We delete all columns/attributes not appearing in the SELECT clause. Although we associate this step with the SELECT clause, database theorists call it a “project” or “projection” operation (as in projection of a three-dimensional image to a two-dimensional image).
- 7) Finally, if the query includes DISTINCT, we delete any rows that are exact duplicates of some preceding row after the dropping of columns in the previous step.

Figure 10.4 shows a graphical depiction of this sequence of operations.



**Figure 10.4** Conceptual order of query processing steps.

While the above steps are the conceptual sequence of operations, the query optimizer module of the database system is free to find a more efficient way to compute the same result. The effectiveness of query optimization may vary depending on the complexity of your query and which database software you are using.

One immediate consequence of this conceptual sequence is that it is not possible to use an aggregation function such as `Sum` within a WHERE condition or in the ON condition of a join, because joins and WHERE occur before aggregation. If you need to perform such operations, then your query requires query nesting or chaining, both of which are described in the next chapter.

Another consequence of the conceptual ordering of queries is that expressions appearing in ORDER BY are subject to the same rules as those appearing in SELECT, because queries evaluate both kinds of expressions after aggregation. Specifically, if a query uses GROUP BY, then every attribute appearing in SELECT or ORDER BY must either be grouped by or should be within an aggregation function. For example, consider the hypothetical query:

```

SELECT    FirstName, LastName,
          Sum(UnitPrice*Quantity) AS Revenue
FROM      CUSTOMER, Orders, ORDERDETAIL, PRODUCT
WHERE     CUSTOMER.CustomerID = Orders.CustomerID AND
          Orders.OrderID = ORDERDETAIL.OrderID AND
          ORDERDETAIL.ProductID = PRODUCT.ProductID
GROUP BY  CUSTOMER.CustomerID, FirstName, LastName
ORDER BY  ProductName;                                (Incorrect)

```

This query will generate an error message because we are not grouping by *ProductName*. Fundamentally, this query makes no sense, because we cannot sort the grouped output by the value of an attribute that can take many different values within each group: each order may contain more than one product, so there is often no single appropriate product name to associate with a particular order. Therefore, we cannot use *ProductName* in the SELECT or ORDER BY clauses unless it appears within an aggregation function.

## Exercises

- 10.1** (Queries of a single table without filtering conditions) Based on the plumbing supply store database from Chapter 7, write SQL SELECT queries that display the following information:
- A** The name of each product.
  - B** All days on which orders have been placed, with no duplicate entries in the case of more than one order placed on the same day.
  - C** The entire PRODUCT table.
  - D** For each product, the product ID, product name, and the ratio of the number of units on order to the number of units in stock.
- 10.2** (Queries of a single table without filtering conditions) Based on the conference database `conference.mdb` on the book website, write SQL SELECT queries to display the following information:
- A** The first and last name of each speaker.
  - B** A list of all areas of expertise claimed by speakers, with no repetition in the case of multiple speakers with the same expertise area.
  - C** The entirety of the rooms table.
- 10.3** (Queries of a single table with filtering conditions) Based on the plumbing supply store database from Chapter 7, write SQL queries that perform the following tasks:
- A** Show the name and unit price of all products priced below \$50 (note: SQL treats currency amounts like any other number, so you should *not* use a \$ sign or quotes in your query.)

- B** Show the product name, units on order, and units in stock for all products for which the number of units on order is at least 40% of the number of units in stock.
  - C** Show the same information as in part (b), but with the additional restriction that the number of units on order is no more than 10.
  - D** Show the first name, last name, city, and state of all customers outside New Jersey (state code “NJ”).
  - E** Show the first name, last name, city, and state of all customers who are outside New Jersey or have the first name “Robert” (or both).
- 10.4** (Queries of a single table with filtering conditions) Based on the conference database on the website, write SQL SELECT queries to display the following information:
  - A** The titles, dates, and start times of all sessions in room “101”; note that the *RoomID* field in this database is text, not a number.
  - B** The room IDs, capacities, and notes for all rooms with a capacity below 100.
  - C** The same information as in part (b), but only for rooms capable of serving refreshments; note that in SQL the possible values of a yes/no field are True and False (without quotes).
- 10.5** (Simple inner joins) Based on the plumbing supply store database from Chapter 7, write SQL queries that use an INNER JOIN clause to perform the following tasks:
  - A** Show the entirety of each combination of a record from the ORDERDETAIL table and a record from the PRODUCT table for which the *ProductID* fields match.
  - B** For the same combinations of ORDERDETAIL and PRODUCT records as in part (a), show the entirety of the ORDERDETAIL record, but only the product name and unit price fields from the PRODUCT record.
  - C** Show the same information as in part (b), but only for items costing at least \$1,000 (again, note that you do not use \$ signs in front of currency amounts in SQL).
  - D** Produce the same output as in part (c), but with an additional column showing the unit price of the product multiplied by the quantity ordered.
  - E** For all combinations of a record from the ORDERDETAIL table and a record from the PRODUCT table that have matching *ProductID* fields and also have a quantity ordered that is at least half the number of units now in stock, show the quantity ordered, the product name, and the number of units in stock.

- 10.6** (A Cartesian join) Based on the plumbing supply store database from Chapter 7, write a query that shows the first name, last name, and product name for all possible combinations of a record from the CUSTOMER table and record from the PRODUCT table, without regard to whether the customer has ever ordered the product.
- 10.7** (Inner joins using WHERE) Repeat all the queries from Problem 5, but use a WHERE clause instead of an INNER JOIN clause.
- 10.8** (Compound joins using WHERE) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:
- A** Write a query that shows, for each row of the ORDERDETAIL table, the first and last name of the corresponding customer, the date of the order, the quantity ordered, and the name of the product. Use only WHERE conditions to join the tables.
  - B** Create a query producing the same output as part (a) using MS Access Query Design View. Switch to SQL view and compare the contents of the FROM clause to what you wrote in part (a).
  - C** Perform the same query as part (a), using WHERE to join tables, but only include rows for which the quantity ordered is at least 5.
  - D** Repeat the query from part (a), but showing all rows for which the quantity ordered is at least 5 or the unit price is over \$1,500.
- 10.9** (More compounds joins) Based on the conference database on the website, and using WHERE to express joins between tables, write SQL queries for the following tasks. A “presentation” consists of a single speaker appearing in a single session.
- A** Show the title, starting time, room ID, and room capacity for each session.
  - B** For each presentation at the conference, show the speaker first name, speaker last name, speaker area of expertise, session title, room ID, and session start time.
  - C** Show the same information as part (b), but only for speakers from Florida (state code “FL”).
  - D** Show the same information as part (b), but only for speakers who are from Florida or whose area of expertise is “Wellness.”
  - E** For each presentation being given in a room with a capacity of at least 100, show the speaker ID, speaker first name, speaker last name, room ID, and room capacity.
  - F** Show the speaker ID, first name, and last name for all speakers giving any presentations in a room with a seating capacity of at least 100.

- 10.10** (Simple aggregation without grouping) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:
- A** Show the total number of units of held in stock (summed across all products).
  - B** Show the total value of inventory held, with each unit of inventory valued at its unit price.
  - C** Show the total value of inventory held in products whose price is below \$50.
  - D** Show the total value of inventory held in products whose price is between \$100 and \$750 (inclusive).
  - E** Show the number of products whose unit price is under \$200.
- 10.11** (More simple aggregation without grouping) Based on the conference database on the website, and using WHERE to express joins between tables, write SQL queries to display the following information. A “presentation” consists of a single speaker appearing in a single session.
- A** The total combined seating capacity of all rooms.
  - B** The smallest room seating capacity, the average room seating capacity, and the largest room seating capacity.
  - C** The number of speakers attending the conference.
  - D** The number of presentations being given at the conference.
  - E** The number of presentations being given in rooms whose capacity is below 100.
  - F** The number of sessions taking in place in rooms that can serve refreshments.
- 10.12** (Aggregation with grouping) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:
- A** For each customer, show the Customer ID, first name, last name, and the total number of orders placed.
  - B** Show the same information about customers as in part (a), but count only orders placed since April 28, 2013; if a customer has placed no orders since that time, they need not appear in the output.
  - C** Show the same information as in part (b), but also include the date of each customer’s most recent order.
  - D** For each product, show its name and average quantity ordered when it appears in orders placed by customers from New Jersey (state code “NJ”).
- 10.13** (More aggregation with grouping) Based on the conference database on the website, and using WHERE to express joins between tables, write SQL queries for the following tasks:



- A For each room in which any session is being held, show the room ID and the number of sessions being held there (with the column heading *NumSessions*).
- B Show the same information as part (a), but also show the capacity of each room.
- C For each room, show its ID, capacity, and the number of *presentations* being given there (the heading for this column should be *NumPresentations*).
- D For each speaker, show their first name, last name, the number of presentations they are giving (with the column heading *NumPresentations*). Make sure that if the conference were large enough to have two speakers with the same name, each would appear on a different line of the output.

**10.14** (More aggregation with grouping) Download, save, and open the database *bookstore-2000.mdb* from the website. This database is a somewhat more complicated retail database than our plumbing database. The price actually charged the customer is the *cost\_of\_each* attribute in the ORDERLINES table, and can be lower than the *retail\_price* attribute in the BOOKS table, reflecting occasional discounts. The database contains a small amount of redundancy, in that *cost\_line* in the ORDERLINES table is always equal to *quantity\*cost\_each*. Using WHERE to express joins between tables, write SQL queries to for the following tasks:

- A Show the name of each author, the number of books written by the author (labeled *numb\_books*), the most recent year of publication for such books (labeled *most\_recent*), and their average retail price (labeled *avg\_price*).
- B For each combination of publisher and year after 1985 for which any books were published, show the publisher name, the year, and the number of books published by that publisher in that year (labeled *numb\_books*).
- C For each customer, show their customer number, first name, last name, and the total dollar value of discounts they have received; note that the total discounts received is equal to the sum of *quantity\*(retail\_price - cost\_each)*, or equivalently the sum of *quantity\*retail\_price - cost\_line*. Note that the total discounts received are zero for most customers. The discounts column should have the heading *discounts*.
- D For each publisher, show its name and the total value of inventory on hand from that publisher, which is the total of *retail\_price\*number\_on\_hand* for each book by that publisher. This column should have the label *inventory\_value*.

- 10.15** (Criteria after aggregation) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:
- A** Show the name and total revenue for each product from which you have at least \$2,000 in revenue.
  - B** Show the name and total revenue between April 22 and 27, 2013 (inclusive) for each product for which you have at least \$1,500 in revenue between those dates.
  - C** Show the first name, last name, and number of orders for all customers who have placed at least 3 orders.
  - D** Show the first name, last name, and number of orders placed on or after April 25, 2013, for all customers who have at least 2 such orders.
- 10.16** (Criteria after aggregation) Based on the conference database on the website, write SQL queries to for the following tasks:
- A** Show the first and last name of all speakers who are appearing in least three sessions. Also show the number of sessions each such speaker is appearing in, labeled *NumSessions*.
  - B** Show the first and last name of all speakers who are appearing at least two sessions being held in rooms with a large screen. Also show the number of such sessions, labeled *NumLargeScreen*.
  - C** Show the ID and capacity for each room in which at least three sessions are being held. Also show the number of sessions being held in each of these rooms, labeled *NumSessions*.
- 10.17** (Simple sorting) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:
- A** Show the name, unit price, units in stock, and units on order for each product, sorted alphabetically by product name.
  - B** Show the same information as in part (a), but sorted from the largest number of units in stock to the smallest. For products with the same number of units in stock, sort the rows of output from the largest number of units on order to the smallest.
  - C** Show the same information as in parts (a) and (b), and also the *inventory position*, which is the sum of the units in stock and the units on order. Sort the output from largest to smallest inventory position.
  - D** Show the same information as in part (c), and in the same order, but only for products with a unit price of at least \$1,000.
- 10.18** (Sorting information extracted from multiple tables) Based on the plumbing supply store database from Chapter 7, write SQL queries to perform the following tasks:

- A** Show the name of each product and the total amount of revenue it has generated (labeled *Revenue*). The output should be sorted by revenue, with the highest-revenue products first; products that have never been ordered need not appear in the output.
- B** Show the same information as in part (a), but only for items that have generated at least \$1,000 of revenue.
- C** Show the same information as part (a), but including only revenue from orders placed April 27, 2013, or later; products not ordered since that date need not appear in the output.
- D** Show the same information as part (a), but only for items for which at least four total units have been ordered, and also showing the total number of units ordered for each product (labeled *UnitsOrdered*). Sort the output from the largest to the smallest total number of units ordered; within products with the same total number of units ordered, sort from the largest to smallest total revenue.

**10.19** (Mixed queries on the bookstore database) Download the `bookstore-2000.mdb` sample database from the textbook website. Write SQL queries to show the following:

- A** Show the first name, last name, and e-mail of each customer in New Jersey.
- B** For each order, list the customer first and last name, customer phone, and order date.
- C** Repeat the query from part (b), but only show orders placed on or after July 1, 2000.
- D** Show all book titles ordered on or after March 1, 2000. Do not list any title more than once.
- E** Show the title, author name, publisher name, publication year, and number on hand for all books of which there are at least 8 copies on hand, and which were published by Knopf or published after 1980 (or both). Sort the output by the number of copies on hand, with the largest number of books on hand coming first. Books with the same number of copies on hand should appear in alphabetical order by title.
- F** For each author with any books with a publication year before 1950, give the author's name and the number of such books. Sort the results in order of the number of books, with the largest number of books appearing first. The number of books column should have the heading *number\_of\_books*.
- G** Show the number of titles and total number of copies on hand of books that have a retail price less than \$20. These results should respectively be labeled *number\_of\_titles* and *number\_of\_copies*.

- H Compute the total value of current inventory (based on retail prices), labeled as *total\_inventory\_value*.
- I Show the title, author, and number of copies of each book ordered (labeled *number\_sold*) in the period January–June 2000. Books not ordered during this period need not appear. Sort the result alphabetically by author name, with books having the same author sorted alphabetically by title.
- J Show the first name, last name, and number of physical books ordered (labeled *num\_ordered*) for all customers who have ordered at least 10 physical books. Sort the results alphabetically by last name, with customers having the same last name sorted alphabetically by first name.
- K Show the first name and last name of each customer who has spent at least \$45 on books published in 1980 or later. Also show the total amount of such spending. Sort the results from the most spent to the least. Among customers whose spending is identical, display them alphabetically by last name.

**10.20** (Mixed queries on the conference database) From the textbook website, download the conference database. Write SQL queries for the following:

- A Show a “master schedule” for the conference. For each presentation at the conference, this query should list the date, session start time, session title, room, speaker first name, and speaker last name. It should be sorted by date, then by session start time, then by session title, then by speaker last name.
- B Show the ID, name, date, and start time of each session, along with the number of speakers scheduled for the session. Sort the results from the largest number of speakers to the smallest.
- C Show the ID, name, date, and start time of each session, along with the number of speakers scheduled, for each session with at least two speakers. Sort the results from the largest number of speakers to the smallest.
- D Show a schedule for room 101. For each presentation at the conference in room 101, show the session start date, session start time, session title, speaker first name, and speaker last name. Sort the results by order of occurrence (earliest first) and then by speaker last name.
- E Show all rooms with capacity of at least 75 people in which at least 6 presentations are being given. Show the room ID, the room capacity, whether or not the room has a large screen, and the number of presentations. Sort the results by the number of presentations, with the largest number of presentations first. The number-of-presentations column should have the heading *NumberOfPresentations*.

- F** Show the average capacity of rooms that either have a large screen or have the ability to serve refreshments (or both). The result of this query should be a single cell.
- G** For each session, show the title of the session and a single count of the number of speakers in the session whose expertise is either “Student Life” or “Residence Halls.” If there are no such speakers in a given session, then that session need not appear in the query output.
- H** For each room, show the room ID and the number of talks being given in the room by speakers from Pennsylvania (“PA”) or Georgia (“GA”). Display only one count per room, not separate counts for the two states. Rooms with no such talks need not appear in the output. Sort the results by the number of talks, with the largest number first. The column heading for the number-of-talks column should be *PAGATalks*.