Retrieving Data from a MySQL Database

- One of the most important functions that a relational database management system (RDBMS) must support is the ability to access data in the databases managed by that system.
- Data access must extend beyond the mere retrieval of information as it is stored in the tables.
- You must be able to choose which data you want to view and how that data is displayed.
- To support this functionality, MySQL provides an SQL statement that is both powerful and flexible in its implementation.
- The SELECT statement is the primary SQL statement used in MySQL—and in most RDBMSs—to retrieve specific data from one or more tables in a relational database.

SELECT Statement

- By using a SELECT statement, you can specify which columns and which rows to retrieve from one or more tables in your MySQL database.
- You can also link values together across multiple tables, perform calculations on those values, or group values together in meaningful ways in order to provide summarized information.
- When you execute a SELECT statement, the values returned by that statement are presented in the form of a result set, which is an unnamed temporary table that contains the information retrieved from the tables.
- In this lecture, we will discuss how to create SELECT statements that allow you to retrieve exactly the information that you need.

SELECT Statement Cont'd

- Whenever you want to retrieve data from a MySQL database, you can issue a SELECT statement that specifies what data you want to have returned and in what manner that data should be returned.
- For example, you can specify that only specific columns or rows be returned.
- You can also order the rows based on the values in one or more columns.
- In addition, you can group together rows based on repeated values in a column in order to summarize data.
- The SELECT statement is one of the most powerful SQL statements in MySQL.
- It provides a great deal of flexibility and allows you to create queries that are as simple or as complex as you need to make them.
- The syntax for a SELECT statement is made up of a number of clauses and other elements, most of which are optional, that allow you to refine your query so that it returns only the information that you're looking for.

```
<select statement>::=
SELECT
[<select option> [<select option>...]]
{* | <select list>}
[<export definition>]
[
   FROM <table reference> [{, <table reference>}...]
   [WHERE <expression> [{<operator> <expression>}...]]
   [GROUP BY <group by definition>]
   [HAVING <expression> [{<operator> <expression>}...]]
   [ORDER BY <order by definition>]
   [LIMIT [<offset>,] <row count>]
   [PROCEDURE <procedure name> [(<argument> [{, <argument>}...])]]
   [{FOR UPDATE} | {LOCK IN SHARE MODE}]
]

<select option>::=
{ALL | DISTINCT | DISTINCTROW}
| HIGH_PRIORITY
| {SQL_BIG_RESULT | SQL_SMALL_RESULT}
| SQL_BUFFER_RESULT
| {SQL_CACHE | SQL_NO_CACHE}
| SQL_CALC_FOUND_ROWS
| STRAIGHT_JOIN
```

```
<select list>::=
{<column name> | <expression>} [[AS] <alias>]
[{, {<column name> | <expression>} [[AS] <alias>]}...]

<export definition>::=
INTO OUTFILE '<filename>' [<export option> [<export option>]]
| INTO DUMPFILE '<filename>'

<export option>::=
{FIELDS
   [TERMINATED BY '<value>']
   [[OPTIONALLY] ENCLOSED BY '<value>']
   [ESCAPED BY '<value>']}
| {LINES
  [STARTING BY '<value>']
   [TERMINATED BY '<value>']}

<table reference>::=
<table name> [[AS] <alias>]
[{USE | IGNORE | FORCE} INDEX <index name> [{, <index name>}...]]

<group by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
[WITH ROLLUP]

<order by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
```

SELECT statement can contain a number of elements

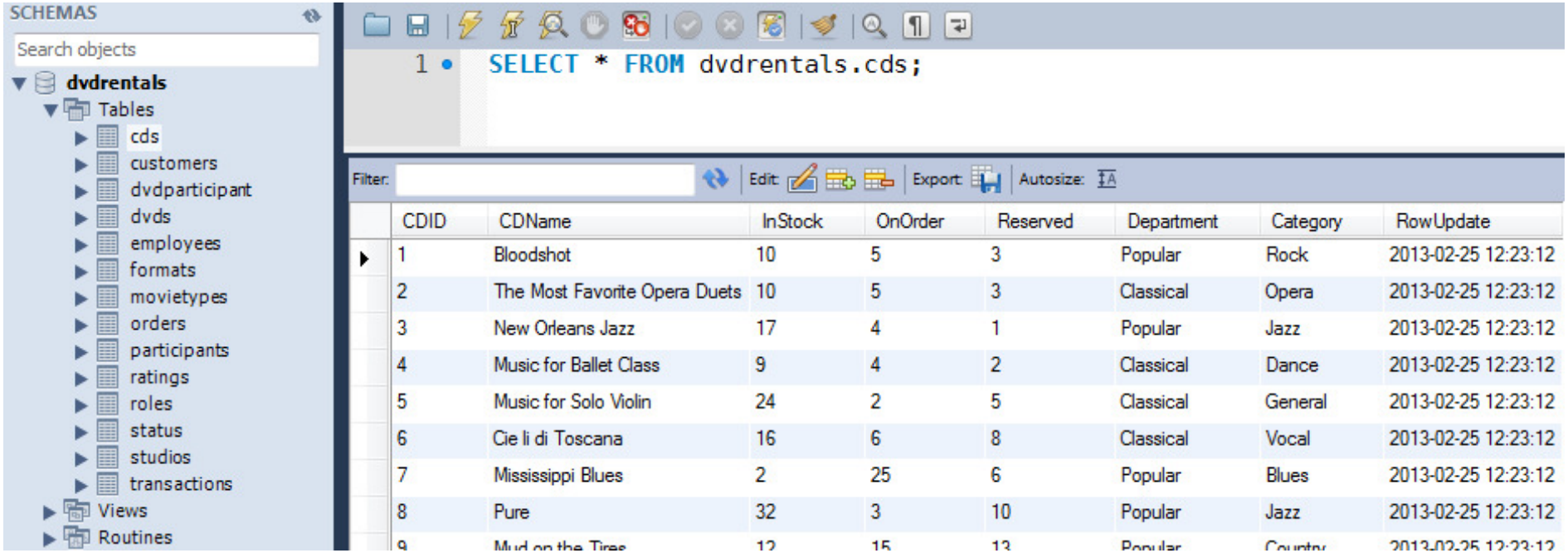
- For most of these elements, the lecture discusses each one in detail, providing the necessary examples to illustrate how they work; however, some elements are covered in later lectures.
- Referring back to the syntax, a SELECT syntax requires only the following clause:

  SELECT {* | <select list>}
- The SELECT clause includes the SELECT keyword and an asterisk (*) or the select list, which is made up of columns or expressions, as shown in the following syntax:

  <select list>::=
  {<column name> | <expression>} [[AS] <alias>]
  [{, {<column name> | <expression>} [[AS] <alias>]}...]
- The select list must include at least one column name or one expression
- If more than one column is included, they must be separated by commas.
- In addition, you can assign an alias to a column name by using the AS subclause.
- That alias can be used in other clauses in the SELECT statement; but, it cannot be used in a WHERE clause because of the way MySQL processes a SELECT statement.

The examples are all based on a table named CDs, which is shown in the following table definition:

```
CREATE TABLE CDs
(
    CDID SMALLINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    CDName VARCHAR(50) NOT NULL,
    InStock SMALLINT UNSIGNED NOT NULL,
    OnOrder SMALLINT UNSIGNED NOT NULL,
    Reserved SMALLINT UNSIGNED NOT NULL,
    Department ENUM('Classical', 'Popular') NOT NULL,
    Category VARCHAR(20) NOT NULL,
    RowUpdate TIMESTAMP NOT NULL
);
```

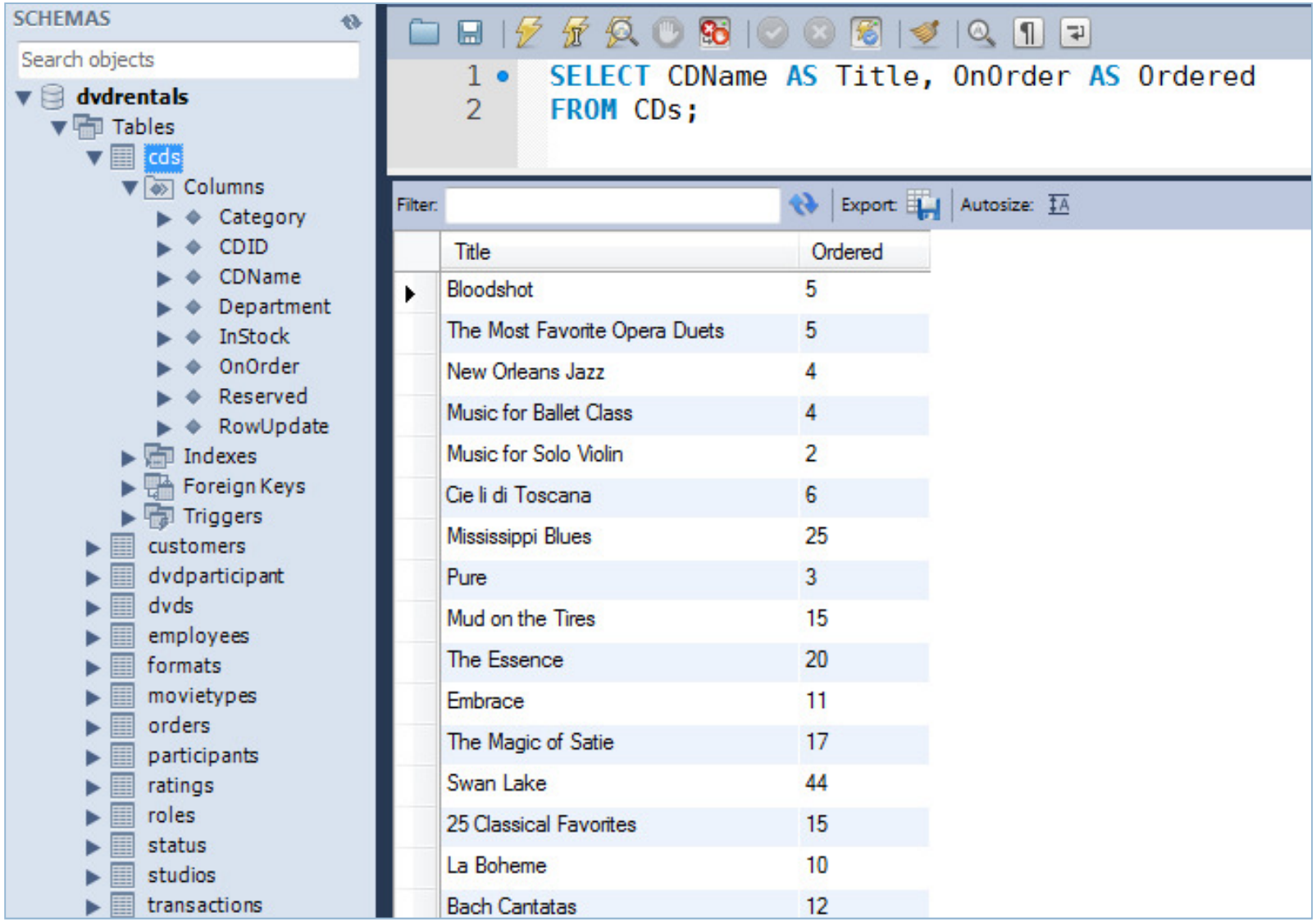

SCHEMAS
Search objects
dvdrentals
Tables
cds
customers
dvdparticipant
dvds
employees
formats
movietypes
orders
participants
ratings
roles
status
studios
transactions
Views
Routines
1 • SELECT * FROM dvdrentals.cds;
Filter:
Edit:
Export:
Autosize:
CDID | CDName | InStock | OnOrder | Reserved | Department | Category | RowUpdate
1 | Bloodshot | 10 | 5 | 3 | Popular | Rock | 2013-02-25 12:23:12
2 | The Most Favorite Opera Duets | 10 | 5 | 3 | Classical | Opera | 2013-02-25 12:23:12
3 | New Orleans Jazz | 17 | 4 | 1 | Popular | Jazz | 2013-02-25 12:23:12
4 | Music for Ballet Class | 9 | 4 | 2 | Classical | Dance | 2013-02-25 12:23:12
5 | Music for Solo Violin | 24 | 2 | 5 | Classical | General | 2013-02-25 12:23:12
6 | Cie li di Toscana | 16 | 6 | 8 | Classical | Vocal | 2013-02-25 12:23:12
7 | Mississippi Blues | 2 | 25 | 6 | Popular | Blues | 2013-02-25 12:23:12
8 | Pure | 32 | 3 | 10 | Popular | Jazz | 2013-02-25 12:23:12

Although using an asterisk in the SELECT clause is an easy way to retrieve every column from a table, it is not a recommended method to use when embedding a SELECT statement in a programming language. Columns can change or be added or deleted from a table. Consequently, unless you're simply performing an ad hoc query and want to view a table's contents quickly, you should normally specify the column names, as shown in the following SELECT statement:

```
SELECT CDID, CDName, Category
FROM CDs;
```

Notice that, in this case, the query specifies three column names in the SELECT clause: CDID, CDName, and Category. Because these names are specified, only data from these three columns is returned by your query, as shown in the following results:

CDID	CDName	Category
1	Bloodshot	Rock
2	The Most Favorite Opera Duets	Opera
3	New Orleans Jazz	Jazz
4	Music for Ballet Class	Dance
5	Music for Solo Violin	General
6	Cie li di Toscana	Vocal
7	Mississippi Blues	Blues
8	Pure	Jazz
9	Mud on the Tires	Country
10	The Essence	New Age
11	Embrace	New Age
12	The Magic of Satie	General
13	Swan Lake	Dance
14	25 Classical Favorites	General
15	La Boheme	Opera

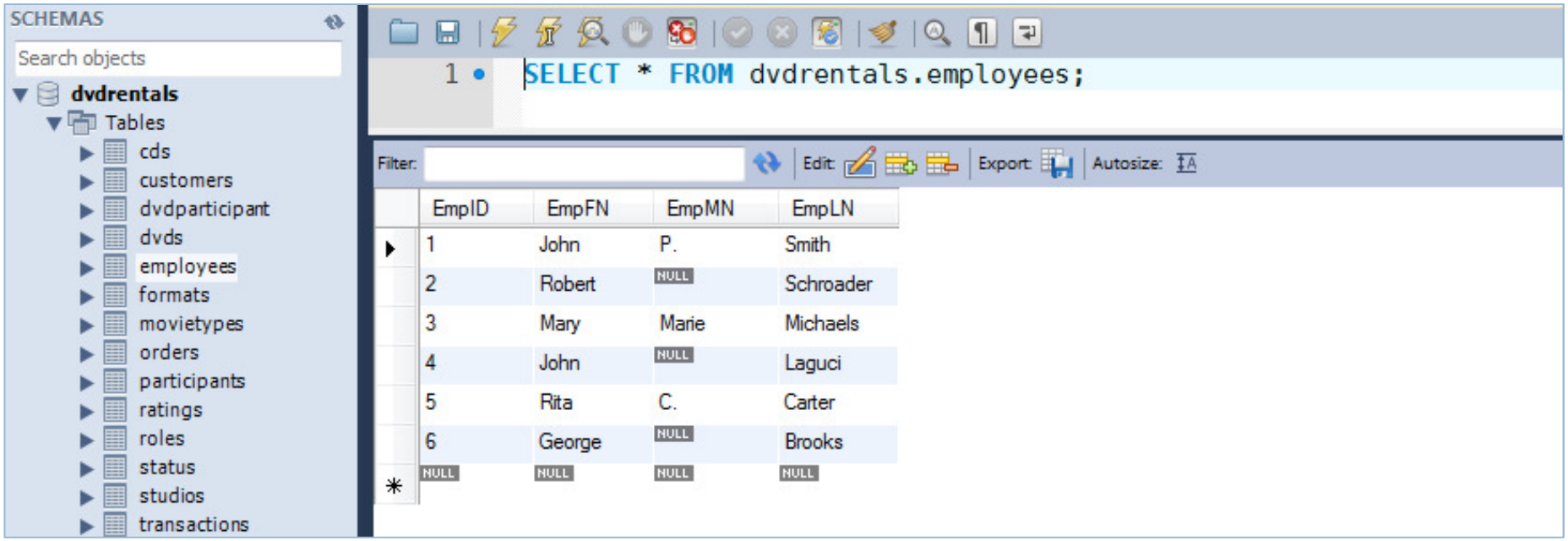


In a situation like this, in which column names are used rather than expressions and the column names are short and simple, supplying an alias isn't particularly beneficial.

As you add expressions to your SELECT clause, create join conditions, or decide to clarify column names, aliases become very useful.

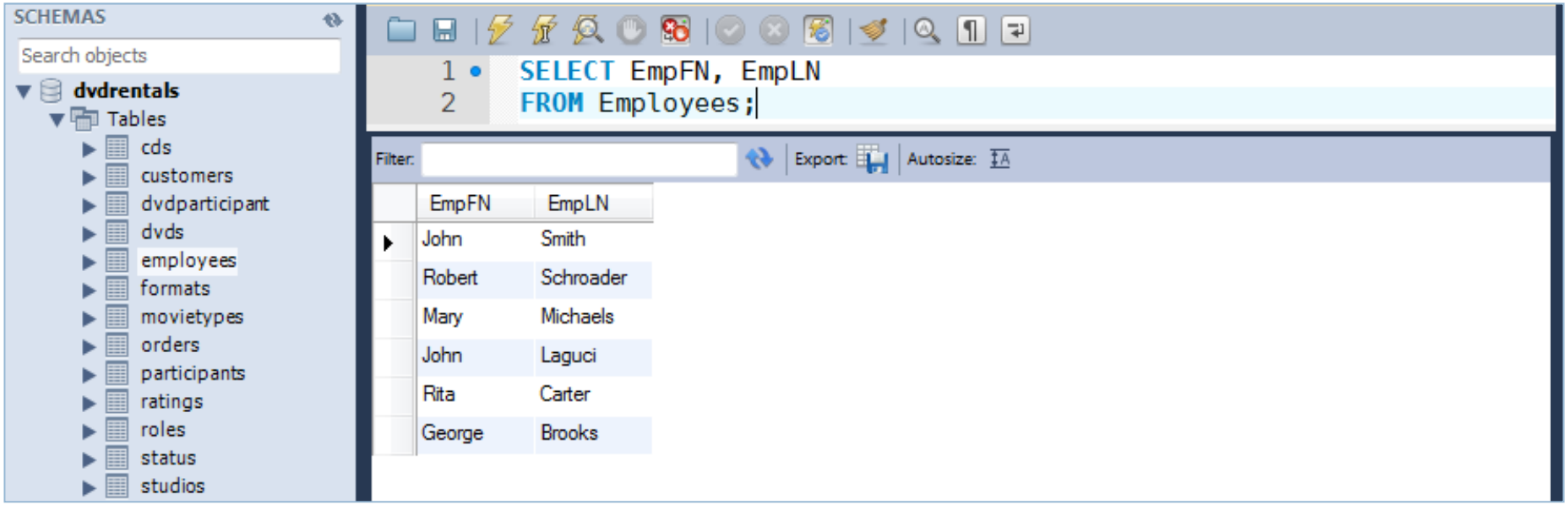
Adding clauses to SELECT statement

- Once you've mastered the SELECT clause and the FROM clause, you can create a basic SELECT statement to query data in any table.
- In most cases, however, you want to limit the number of rows returned and control how data is displayed.
- For this, you need to add more clauses.
- These clauses, which are explained throughout the rest of the lectures, must be added to your statement in the order they are listed in the syntax.
- MySQL processes the clauses in a SELECT statement in a very specific order, so you must be aware of how clauses are defined in order to receive the results that you expect.

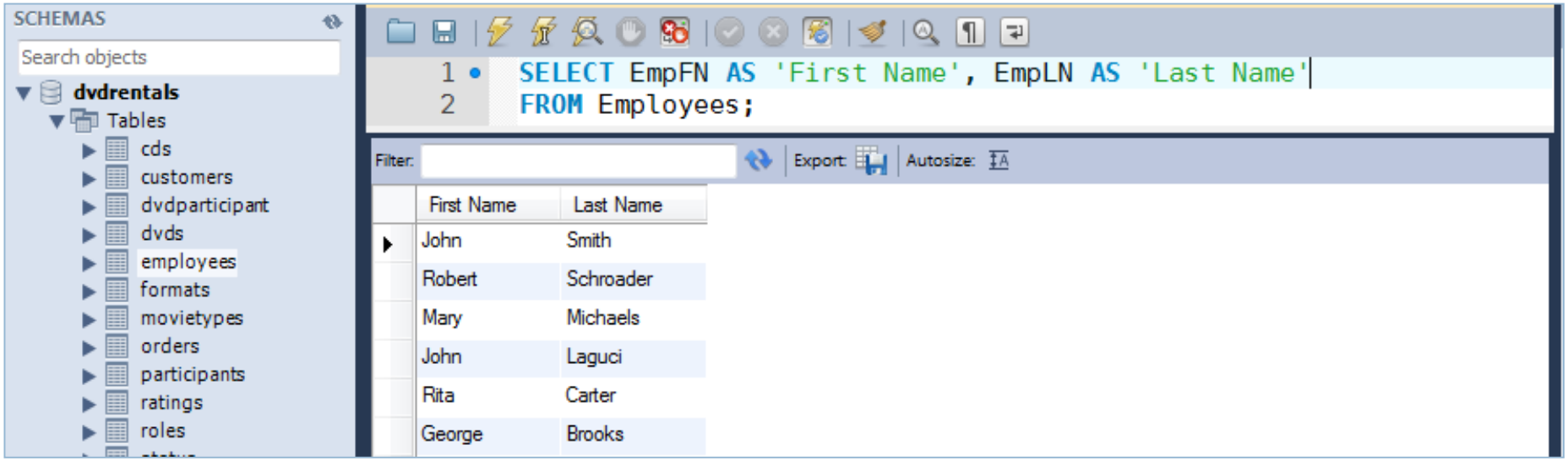


The first SELECT statement that you create retrieves all columns and all records from the Employees table.

Next, retrieve values only from the EmpFN and EmpLN columns of the Employees table.



Now retrieve values from the same columns as the last step, but this time provide aliases for those columns.


SCHEMAS
Search objects
dvdrentals
Tables
cds
customers
dvdparticipant
dvds
employees
formats
movietypes
orders
participants
ratings
roles
SELECT EmpFN AS 'First Name', EmpLN AS 'Last Name'
FROM Employees;
Filter:
Export:
Autosize:
First Name
Last Name
John
Smith
Robert
Schroader
Mary
Michaels
John
Laguci
Rita
Carter
George
Brooks

Whenever you create a SELECT statement that retrieves data from a table in a MySQL database, you must, at the very least, include a SELECT clause and a FROM clause. The SELECT clause determines which columns of values are returned and the FROM clause determines from which tables the data is retrieved. For example, the first SELECT statement that you created retrieves all columns from the Employees table, as shown in the following statement:

```
SELECT * FROM Employees;
```

This statement uses an asterisk to indicate that all columns should be retrieved. Your second SELECT statement specified which columns of data should be returned:

```
SELECT EmpFN, EmpLN
FROM Employees;
```

In this case, the query returns only values in the EmpFN and EmpLN columns because those are the columns specified in the SELECT clause. And as with the first SELECT statement in this exercise, the values were retrieved from the Employees table because that is the table specified in the FROM clause.

The last SELECT statement that you created in this exercise assigns aliases to the columns names, as shown in the following statement:

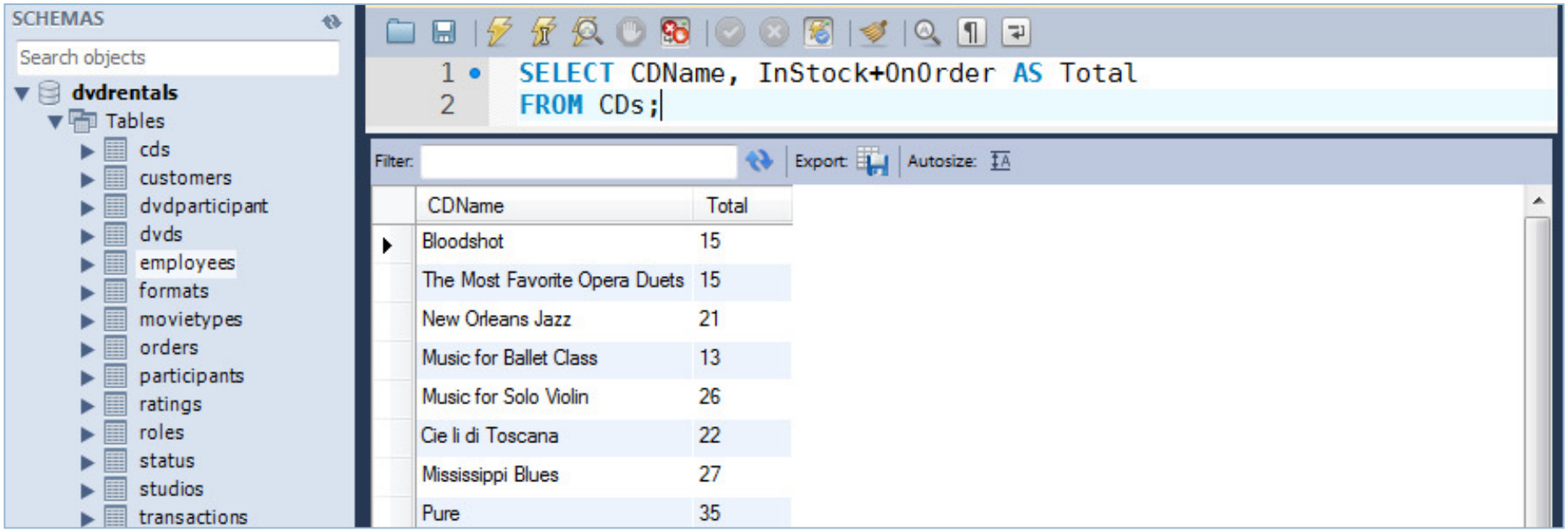
```
SELECT EmpFN AS 'First Name', EmpLN AS 'Last Name'
FROM Employees;
```

Using Expressions in a SELECT Statement

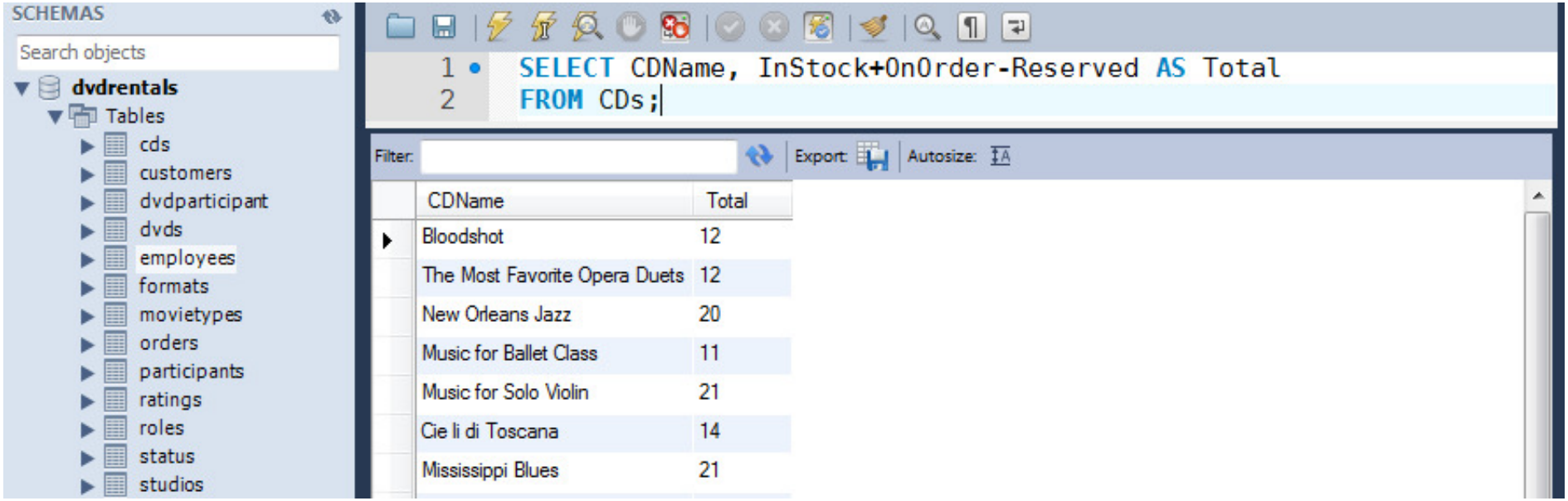
- Recalling from the select list syntax, your select list can include column names or expressions.
- Up to this point, the example SELECT statements that you've seen have included columns names.
- Expressions are also very useful in creating robust SELECT statements that can return the data necessary to your applications.
- An expression is a type of formula that helps define the value that the SELECT statement will return.
- An expression can include column names, literal values, operators, and functions.
- An operator is a symbol that represents the action that should be taken, such as comparing values or adding values together.

- Now take a look at a SELECT statement that contains an expression in its select list.
- The following statement retrieves information from several columns in the table:

```
SELECT CDName, InStock+OnOrder AS Total
FROM CDs;
```


SCHEMAS
Search objects
dvdrentals
Tables
cds
customers
dvdparticipant
dvds
employees
formats
movietypes
orders
participants
ratings
roles
status
studios
transactions
1 SELECT CDName, InStock+OnOrder AS Total
2 FROM CDs;
Filter:
Export:
Autosize:
CDName
Total
Bloodshot 15
The Most Favorite Opera Duets 15
New Orleans Jazz 21
Music for Ballet Class 13
Music for Solo Violin 26
Cie li di Toscana 22
Mississippi Blues 27
Pure 35

Creating expressions in your select list that are more complex than the one in the last statement.



Using Variables in a SELECT Statement

One type of expression that you can include in your select list is one that allows you to define a variable. A *variable* is a type of placeholder that holds a value for the duration of a client session. This is useful if you want to reuse a value in later `SELECT` statements.

You define a variable by using the following structure:

```
@<variable name>:={<column name> | <expression>} [[AS] <alias>]
```

The variable name must always be preceded by the at (@) symbol, and the variable value must always be specified by using the colon/equal sign (:=) symbols. In addition, a variable can be associated with only one value, so your `SELECT` statement should return only one value per variable. If your `SELECT` statement returns more than one value for a variable, the last value returned is used by the variable. If you want to define more than one variable in a `SELECT` statement, you must define each one as a separate select list element. For example, the following `SELECT` statement defines two variables:

```
SELECT @dept:=Department, @cat:=Category
FROM CDs
WHERE CDName='Mississippi Blues';
```

When you execute this statement, the values from the Department column and the Category column are stored in the appropriate variables. For example, the row in the CDs table that contains a CDName of Mississippi Blues contains a Department value of Popular and a Category value of Blues. As a result, the Popular value is assigned to the `@dept` variable, and the Blues value is assigned to the `@cat` variable. When you execute the `SELECT` statement, you should receive results similar to the following:

```
+-------------------+----------------+
| @dept:=Department | @cat:=Category |
+-------------------+----------------+
| Popular           | Blues          |
+-------------------+----------------+
1 row in set (0.26 sec)
```

As you can see, your result set displays the values assigned to your variables. Once you've assigned values to your variables, you can then use them in other `SELECT` statements, as shown in the following example:

```
SELECT CDID, CDName, InStock+OnOrder-Reserved AS Available
FROM CDs
WHERE Department=@dept AND Category=@cat;
```

Using a SET statement to define a variable

- In addition to a SELECT statement, you can use a SET statement to define a variable.
- For example, the following SET statement defines the same variables used previously:

```
SET @dept='Popular', @cat='Blues';
```

- In this case, rather than setting the variable values based on values returned by a SELECT statement, you can specify the values directly, as shown here.
- You can then use the variables in subsequent SELECT statements in your client session, as you would variables defined in a SELECT statement.
- In either case, the variables are usable for only as long as the client session lasts.

```
1  SET @dept='Popular', @cat='Blues';
2
3  SELECT CDID, CDName, InStock+OnOrder-Reserved AS Available, @dept AS Dept, @cat AS Category
4  FROM CDs
5  WHERE Department=@dept AND Category=@cat;
```

Filter: Export: Autosize:

CDID	CDName	Available	Dept	Category
7	Mississippi Blues	21	Popular	Blues
19	Richland Woman Blues	20	Popular	Blues
22	Runaway Soul	31	Popular	Blues
23	Stages	34	Popular	Blues

In-class Exercise

```
SELECT @rating:=RatingID
FROM DVDs
WHERE DVDName='White Christmas';

SELECT DVDID, DVDName, MTypeID
FROM DVDs
WHERE RatingID=@rating;
```

Using a SELECT Statement to Display Values

- When we started discussing about the SELECT statement syntax, you may have noticed that nearly all elements of the statement are optional.
- Although your SELECT statements normally include a FROM clause, along with other optional clauses and options, these elements are all considered optional because you can use a SELECT statement to return values that are not based on data in a table.
- When using only the required elements of a SELECT statement, you need to specify only the SELECT keyword and one or more elements of the select list.
- The select list can contain literal values, operators, and functions, but no column names.

- For example, the following SELECT statement includes three select list elements:

```
SELECT 1+3, 'CD Inventory', NOW() AS 'Date/Time';
```

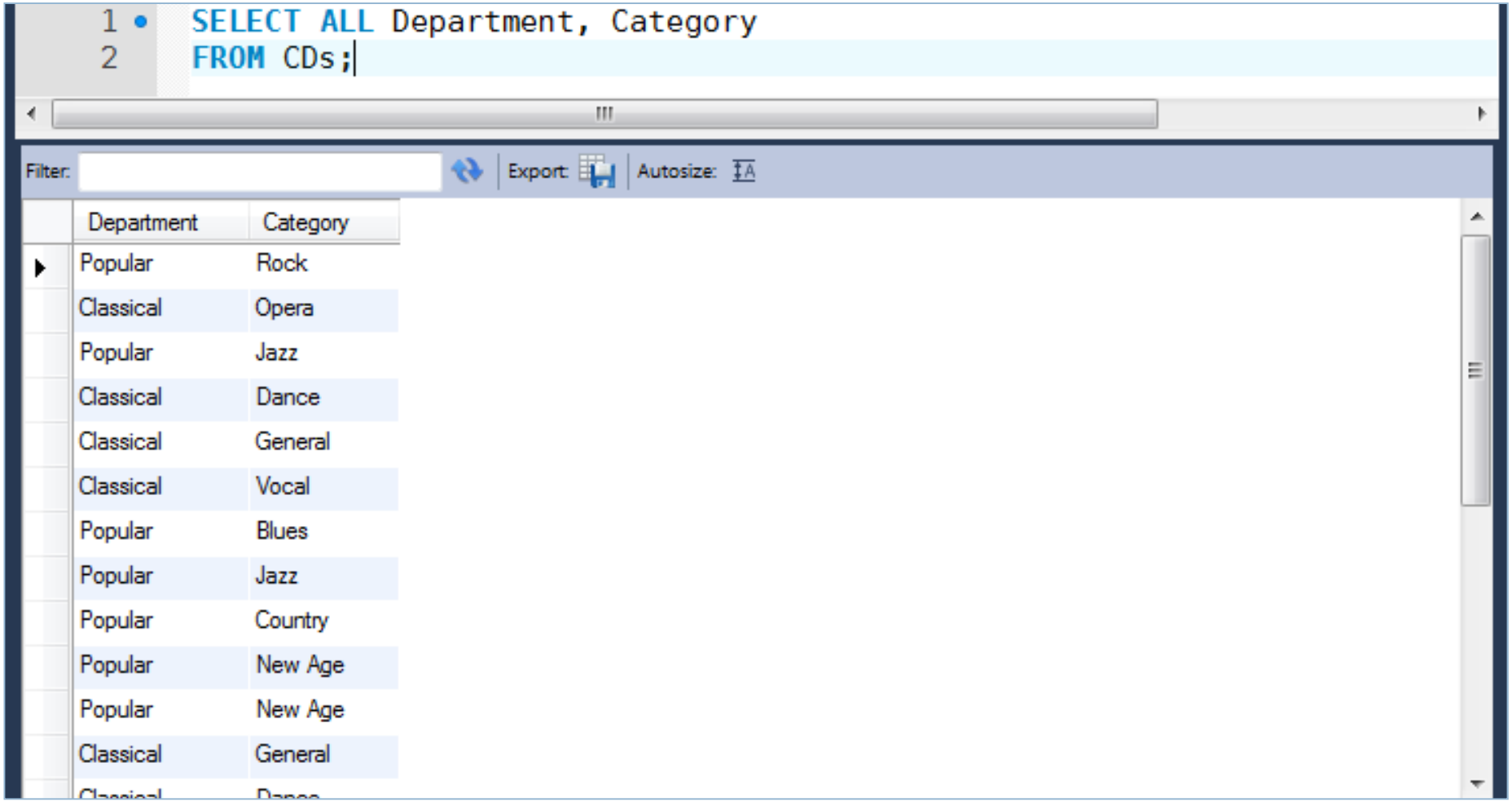
The SELECT Statement Options

When you create a SELECT statement, your SELECT clause can include one or more options that are specified before the select list. The options define how a SELECT statement is processed and, for the most part, how it applies to the statement as a whole, rather to the specific data returned. As the following syntax shows, you can include a number of options in a SELECT statement:

```
<select option>::=
{ALL | DISTINCT | DISTINCTROW}
| HIGH_PRIORITY
| {SQL_BIG_RESULT | SQL_SMALL_RESULT}
| SQL_BUFFER_RESULT
| {SQL_CACHE | SQL_NO_CACHE}
| SQL_CALC_FOUND_ROWS
| STRAIGHT_JOIN
```

The following table describes each of the options that you can include in a SELECT statement.

Option	Description
ALL DISTINCT DISTINCTROW	The `ALL` option specifies that a query should return all rows, even if there are duplicate rows. The `DISTINCT` and `DISTINCTROW` options, which have the same meaning in MySQL, specify that duplicate rows should not be included in the result set. If neither option is specified, `ALL` is assumed.
HIGH_PRIORITY	The `HIGH_PRIORITY` option prioritizes the `SELECT` statement over statements that write data to the target table. Use this option only for `SELECT` statements that you know will execute quickly.
SQL_BIG_RESULT SQL_SMALL_RESULT	The `SQL_BIG_RESULT` option informs the MySQL optimizer that the result set will include a large number of rows, which helps the optimizer to process the query more efficiently. The `SQL_SMALL_RESULT` option informs the MySQL optimizer that the result set will include a small number of rows.
SQL_BUFFER_RESULT	The `SQL_BUFFER_RESULT` option tells MySQL to place the query results in a temporary table in order to release table locks sooner than they would normally be released. This option is particularly useful for large result sets that take a long time to return to the client.
SQL_CACHE SQL_NO_CACHE	The `SQL_CACHE` option tells MySQL to cache the query results if the cache is operating in demand mode. The `SQL_NO_CACHE` option tells MySQL not to cache the query results.
SQL_CALC_FOUND_ROWS	You use the `SQL_CALC_FOUND_ROWS` option in conjunction with the `LIMIT` clause. The option specifies what the row count of a result set would be if the `LIMIT` clause were not used.
STRAIGHT_JOIN	You use the `STRAIGHT_JOIN` option when joining tables in a `SELECT` statement. The option tells the optimizer to join the tables in the order specified in the `FROM` clause. You should use this option to speed up a query if you think that the optimizer is not joining the tables efficiently.



To specify an option in a SELECT statement, you must add it after the SELECT keyword.

This statement uses the ALL option to specify that all rows should be included in the result set, even if there are duplicates.

Eliminating duplicates by using the DISTINCT option

```
1  SELECT DISTINCT Department, Category
2  FROM CDs;
```

Filter: Export: Autosize:

Department	Category
Popular	Rock
Classical	Opera
Popular	Jazz
Classical	Dance
Classical	General
Classical	Vocal
Popular	Blues
Popular	Country
Popular	New Age

Specifying multiple options in your SELECT clause

```
SELECT DISTINCT HIGH_PRIORITY Department, Category
FROM CDs;
```

- The SELECT clause includes the DISTINCT option and the HIGH_PRIORITY option.
- Because the HIGH_PRIORITY option has no impact on the values returned, your result set looks the same as the result set in the previous example.

In-class Exercise

```
SELECT ALL RatingID, StatID
FROM DVDs;

SELECT DISTINCT RatingID, StatID
FROM DVDs;
```

The Optional Clauses of a SELECT Statement

- As you saw earlier, the SELECT statement syntax includes a number of optional clauses that help you define which rows your SELECT statement returns and how those rows display.

- Of particular importance to creating an effective SELECT statement are the
  - WHERE
  - GROUP BY
  - HAVING
  - ORDER BY
  - LIMIT

- As you learned earlier, any of these clauses that you include in your SELECT statement must be defined in the order that they are specified in the syntax.

WHERE Clause

- Earlier you saw how you can use a SELECT clause to identify the columns that a SELECT statement returns and how to use a FROM clause to identify the table from which the data is retrieved.
- WHERE clause allows you to specify which rows are returned by your query.
- The WHERE clause is made up of one or more conditions that define the parameters of the SELECT statement.
- Each condition is an expression that can consist of column names, literal values, operators, and functions.
- The following syntax describes how a WHERE clause is defined:

  WHERE <expression> [{<operator> <expression>}...]

- As you can see, a WHERE clause must contain at least one expression that defines which rows the SELECT statement returns.
- When you specify more than one condition in the WHERE clause, those conditions are connected by an AND or an OR operator.

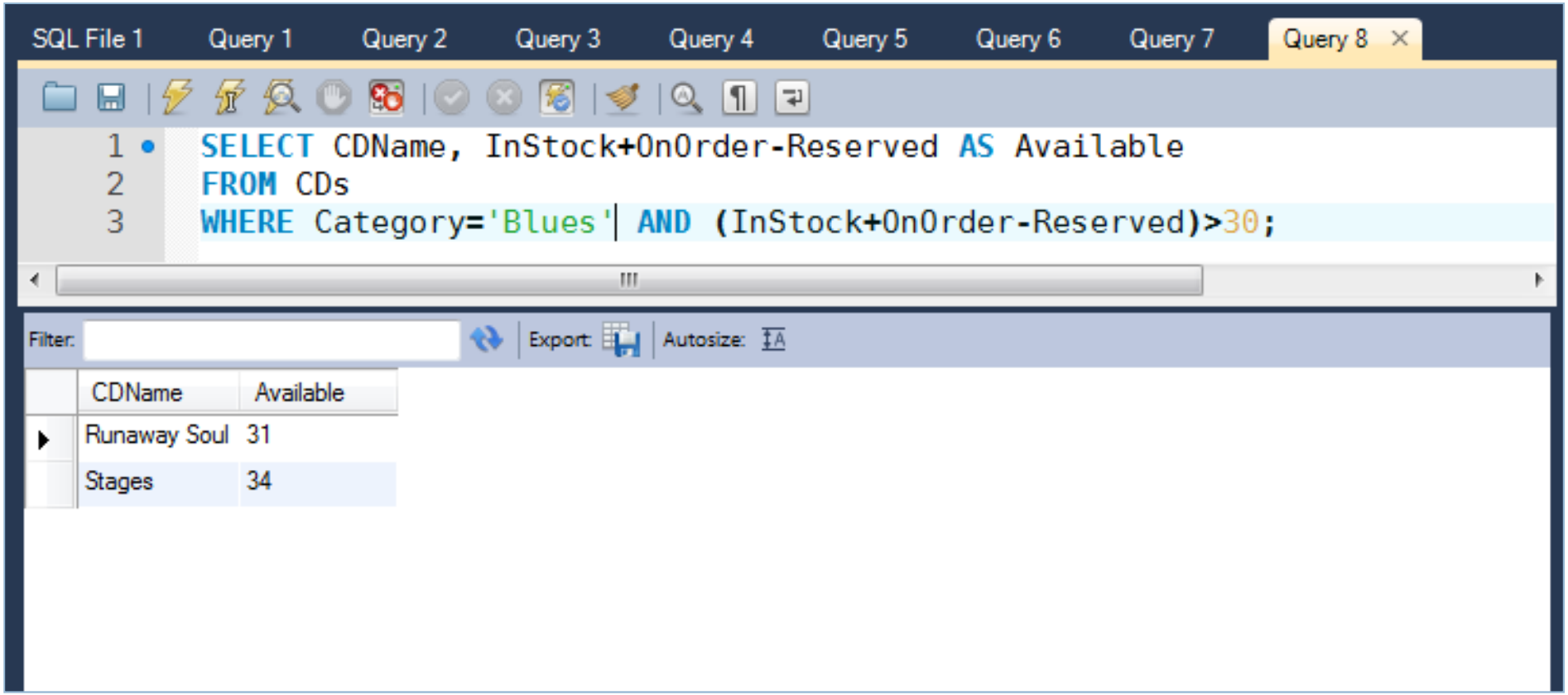
```
1  SELECT CDName, InStock+OnOrder-Reserved AS Available
2  FROM CDs
3  WHERE Category='Blues';
```

Filter: Export: Autosize:

CDName	Available
Mississippi Blues	21
Richland Woman Blues	20
Runaway Soul	31
Stages	34

The WHERE clause indicates that only rows with a Category value of Blues should be returned as part of the result set.

Because the SELECT clause specifies the CDName column and an expression that is assigned the alias Available, only those two columns are included in the result set, as shown in the following results:



Another alternative is to add a HAVING *clause to your* SELECT *statement.*

In that clause, you can use column aliases.

```
1  SELECT CDName, Category, InStock+OnOrder-Reserved AS Available
2  FROM CDs
3  WHERE (Category='Blues' OR Category='Jazz')
4  AND (InStock+OnOrder-Reserved)>20;
```

Filter: Export: Autosize:

CDName	Category	Available
Mississippi Blues	Blues	21
Pure	Jazz	25
Live in Paris	Jazz	28
Runaway Soul	Blues	31
Stages	Blues	34

In-class Exercise

```
SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2';

SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s1' OR StatID='s3' OR StatID='s4';

SELECT DVDName, MTypeID
FROM DVDs
WHERE StatID='s2' AND (RatingID='NR' OR RatingID='G');
```

GROUP BY Clause

- Up to this point, the components of the SELECT statement that you have been introduced to mostly have to do with returning values from columns and rows.
- Even when your SELECT clause included an expression, that expression usually performed some type of operation on the values in a column.
- The GROUP BY clause is a little different from the other elements in the SELECT statement in the way that it is used to group values and summarize information.
- Take a look at the syntax to see how this works:

```
GROUP BY <group by definition>

<group by definition>::=
<column name> [ASC | DESC]
[{, <column name> [ASC | DESC]}...]
[WITH ROLLUP]
```

In order to use a GROUP BY clause effectively, you should also include a select list element that contains a function that summarizes the data returned by the SELECT statement.

For example, suppose you want to know how many compact disk titles are listed in the CDs table for each category.

```
1 •  SELECT Category, COUNT(*) AS Total
2    FROM CDs
3    WHERE Department='Popular'
4    GROUP BY Category;
```

Filter: Export: Autosize:

Category	Total
Blues	4
Country	2
Jazz	3
New Age	2
Rock	1

```
1 • SELECT Department, Category, COUNT(*) AS Total
2   FROM CDs
3   GROUP BY Department, Category;
```

Filter: Export: Autosize:

Department	Category	Total
Classical	Dance	2
Classical	General	6
Classical	Opera	3
Classical	Vocal	1
Popular	Blues	4
Popular	Country	2
Popular	Jazz	3
Popular	New Age	2
Popular	Rock	1

```
1 • SELECT Department, Category, COUNT(*) AS Total
2   FROM CDs
3   GROUP BY Department, Category WITH ROLLUP;
```

Filter: Export: Autosize:

Department	Category	Total
Classical	Dance	2
Classical	General	6
Classical	Opera	3
Classical	Vocal	1
Classical	NULL	12
Popular	Blues	4
Popular	Country	2
Popular	Jazz	3
Popular	New Age	2
Popular	Rock	1
Popular	NULL	12
NULL	NULL	24

WITH ROLLUP

- Notice the several additional rows in the result set.
- For example, the fifth row (the last Classical entry) includes NULL in the Category column and 12 in the Total column.
- The WITH ROLLUP option provides summary data for the first column specified in the GROUP BY clause, as well as the second column.
- As this shows, there are a total of 12 Classical compact disks listed in the CDs table.
- A summarized value is also provided for the Popular department.
  - There are 12 Popular compact disks as well.
- The last row in the result set provides a total for all compact disks.
  - As the Total value shows, there are 24 compact disks in all.

In-class Exercise

```
SELECT OrderID, COUNT(*) AS Transactions
FROM Transactions
GROUP BY OrderID;

SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID;

SELECT MTypeID, RatingID, COUNT(*) AS 'DVD Totals'
FROM DVDs
GROUP BY MTypeID, RatingID WITH ROLLUP;
```

HAVING Clause

- The HAVING clause is very similar to the WHERE clause in that it consists of one or more conditions that define which rows are included in a result set.
- The HAVING clause, though, has a couple of advantages over the WHERE clause.
  - For example, you can include aggregate functions in a HAVING clause.
- An aggregate function is a type of function that summarizes data, such as the COUNT() function. You cannot use aggregate functions in expressions in your WHERE clause.
- In addition, you can use column aliases in a HAVING clause, which you cannot do in a WHERE clause.
- Despite the disadvantages of the WHERE clause, whenever an expression can be defined in either a HAVING clause or a WHERE clause, it is best to use the WHERE clause because of the way that MySQL optimizes queries.
- In general, the HAVING clause is normally best suited to use in conjunction with the GROUP BY clause.

```
HAVING <expression> [{<operator> <expression>}...]
```

```
1  SELECT Category, COUNT(*) AS Total
2  FROM CDs
3  WHERE Department='Popular'
4  GROUP BY Category
5  HAVING Total<3;
```

Filter: Export: Autosize:

Category	Total
Country	2
New Age	2
Rock	1

ORDER BY Clause

- The SELECT statement also includes an ORDER BY clause that allows you to determine the order in which rows are returned in a results set.

- The following syntax describes the elements in an ORDER BY clause:

  ORDER BY <order by definition>
  <order by definition>::=
  <column name> [ASC | DESC]
  [{, <column name> [ASC | DESC]}...]

- As the syntax indicates, the ORDER BY clause must include the ORDER BY keywords and at least one column name.

- You can also specify a column alias in place of the actual name.

- If you include more than one column, a comma must separate them.

```
1  SELECT CDName, InStock, OnOrder
2  FROM CDs
3  WHERE InStock>20
4  ORDER BY CDName DESC;
```

Export: Autosize:

CDName	InStock	OnOrder
The Magic of Satie	42	17
Swan Lake	25	44
Stages	42	0
Richland Woman Blues	22	5
Pure	32	3
Music for Solo Violin	24	2
Morimur (after J. S. Bach)	28	17
Golden Road	23	10

```
1  SELECT Department, Category, CDName
2  FROM CDs
3  WHERE (InStock+OnOrder-Reserved)<15
4  ORDER BY Department DESC, Category ASC;
```

Export: Autosize:

Department	Category	CDName
Popular	Country	Mud on the Tires
Popular	Rock	Bloodshot
Classical	Dance	Music for Ballet Class
Classical	Opera	The Most Favorite Opera Duets
Classical	Vocal	Cie li di Toscana

LIMIT Clause

- The LIMIT clause takes two arguments, as the following syntax shows:

  LIMIT [<offset>,] <row count>

- The first option, <offset>, is optional and indicates where to begin the LIMIT row count.
- If no value is specified, 0 is assumed.
  - (The first row in a result set is considered to be 0, rather than 1.)
- The second argument, <row count> in the LIMIT clause, indicates the number of rows to be returned.
- For example, the following SELECT statement includes a LIMIT clause that specifies a row count of 4.

```
SELECT CDID, CDName, InStock
FROM CDs
WHERE Department='Classical'
ORDER BY CDID DESC
LIMIT 4;
```