<http://www.1keydata.com/sql/sql.html>

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Count**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Another arithmetic function is **COUNT**. This allows us to **COUNT** up the number of row in a certain table. The syntax is,  **SELECT COUNT("column\_name") FROM "table\_name";**  For example, if we want to find the number of store entries in our table,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   we'd key in,  **SELECT COUNT (Store\_Name) FROM Store\_Information;**  Result:   |  | | --- | | **COUNT (Store\_Name)** | | **4** |   **COUNT** and [**DISTINCT**](http://www.1keydata.com/sql/sqldistinct.html) can be used together in a statement to retrieve the number of distinct entries in a table. For example, if we want to find out the number of distinct stores, we'd type,  **SELECT COUNT (DISTINCT Store\_Name) FROM Store\_Information;**  Result:   |  | | --- | | **COUNT (DISTINCT Store\_Name)** | | **3** | |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Group By**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Now we return to the aggregate functions. Remember we used the [**SUM**](http://www.1keydata.com/sql/sql-sum.html) keyword to calculate the total sales for all stores? What if we want to calculate the total sales for *each* store? Well, we need to do two things: First, we need to make sure we **select** the store name as well as total sales. Second, we need to make sure that all the sales figures are **grouped by** stores. The corresponding SQL syntax is,  **SELECT "column\_name1", SUM("column\_name2") FROM "table\_name" GROUP BY "column\_name1";**  Let's illustrate using the following table,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   We want to find total sales for each store. To do so, we would key in,  **SELECT Store\_Name, SUM(Sales) FROM Store\_Information GROUP BY Store\_Name;**  Result:   |  |  | | --- | --- | | **Store\_Name** | **SUM(Sales)** | | **Los Angeles** | **1800** | | **San Diego** | **250** | | **Boston** | **700** |   **GROUP BY Multiple Columns**  In this example, there is only one column associated with **GROUP BY**. It is possible to have two columns or more associated with **GROUP BY**.  The **GROUP BY** keyword is used when we are selecting multiple columns from a table (or tables) and at least one arithmetic operator appears in the [**SELECT**](http://www.1keydata.com/sql/sqlselect.html) statement. Such operators include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**. When that happens, we need to **GROUP BY** all the other selected columns, *i.e.,* all columns except the one(s) operated on by the arithmetic operator. As such, it is important to note that we may have two columns or more associated with **GROUP BY**. The general syntax is as follows:  **SELECT "column\_name1", "column\_name2", ... "column\_nameN", Function("column\_nameN+1") FROM "table\_name" GROUP BY "column\_name1", "column\_name2", ... "column\_nameN";**  **GROUP BY Month / Date / Week**  A common use of the **GROUP BY** function is on a time period, which can be month, week, day, or even hour. This type of query is often combined with the [**ORDER BY**](http://www.1keydata.com/sql/sqlorderby.html) keyword to provide a query result that shows a time series.  For example, to find total daily sales from **Store\_Information**, we use the following SQL:  **SELECT Txn\_Date, SUM(Sales) FROM Store\_Information GROUP BY Txn\_Date;**  Result:   |  |  | | --- | --- | | **Txn\_Date** | **SUM(Sales)** | | **Jan-05-1999** | **1500** | | **Jan-07-1999** | **250** | | **Jan-08-1999** | **1000** | |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **AS**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| In the [**SQL Alias**](http://www.1keydata.com/sql/sqlalias.html) section, we saw that the syntax for using table and column aliases is as follows:  **SELECT "table\_alias"."column\_name1" "column\_alias" FROM "table\_name" "table\_alias";**  The keyword **AS** is used to assign an alias to the column or a table. It is insert between the column name and the column alias or between the table name and the table alias. The syntax for using **AS** is as follows:  **SELECT "table\_alias"."column\_name1" AS "column\_alias" FROM "table\_name" AS "table\_alias";**  Let's take a look at the same example as we used in the **SQL Alias** section. Assume we have the following table, ***Store\_Information***,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   To find total sales by store using **AS** as part of the table and column alias, we type in:  **SELECT A1.Store\_Name Store, SUM(A1.Sales) AS "Total Sales" FROM Store\_Information AS A1 GROUP BY A1.Store\_Name;**  Result:   |  |  |  | | --- | --- | --- | | **Store** |  | **Total Sales** | | **Los Angeles** |  | **1800** | | **San Diego** |  | **250** | | **Boston** |  | **700** | |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Join**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Now we want to look at joins. To do joins correctly in SQL requires many of the elements we have introduced so far. Let's assume that we have the following two tables,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   Table ***Geography***   |  |  | | --- | --- | | **Region\_Name** | **Store\_Name** | | East | Boston | | East | New York | | West | Los Angeles | | West | San Diego |   and we want to find out sales by region. We see that table ***Geography*** includes information on regions and stores, and table ***Store\_Information*** contains sales information for each store. To get the sales information by region, we have to combine the information from the two tables. Examining the two tables, we find that they are linked via the common field, "Store\_Name". We will first present the SQL statement and explain the use of each segment later:  **SELECT A1.Region\_Name REGION, SUM(A2.Sales) SALES FROM Geography A1, Store\_Information A2 WHERE A1.Store\_Name = A2.Store\_Name GROUP BY A1.Region\_Name;**  Result:   |  |  |  | | --- | --- | --- | | **REGION** |  | **SALES** | | **East** |  | **700** | | **West** |  | **2050** |   The first two lines tell SQL to select two fields, the first one is the field "Region\_Name" from table ***Geography*** (aliased as REGION), and the second one is the sum of the field "Sales" from table ***Store\_Information*** (aliased as SALES). Notice how the table aliases are used here: ***Geography*** is aliased as A1, and ***Store\_Information*** is aliased as A2. Without the aliasing, the first line would become  **SELECT Geography.Region\_Name REGION, SUM(Store\_Information.Sales) SALES**  which is much more cumbersome. In essence, table aliases make the entire SQL statement easier to understand, especially when multiple tables are included.  An alternative way to specify a join between tables is to use the **JOIN** and **ON** keywords. In the current example, the SQL query would be,  **SELECT A1.Region\_Name REGION, SUM(A2.Sales) SALES  FROM Geography A1, Store\_Information A2  JOIN Store\_Information A2  ON A1.Store\_Name = A2.Store\_Name  GROUP BY A1.Region\_Name;**  Several different types of joins can be performed in SQL. The key ones are as follows:   * [**Inner Join**](http://www.1keydata.com/sql/inner-join.html) * [**Outer Join**](http://www.1keydata.com/sql/sqlouterjoin.html) * [**Left Outer Join**](http://www.1keydata.com/sql/left-outer-join.html) * [**Cross Join**](http://www.1keydata.com/sql/cross-join.html)   The following sections explain each **JOIN** type in detail. |

- See more at: <http://www.1keydata.com/sql/sqljoins.html#sthash.yPbUbiiC.dpuf>

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Inner Join**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| An inner join in SQL returns rows where there is at least one match on both tables. Let's assume that we have the following two tables,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   Table ***Geography***   |  |  | | --- | --- | | **Region\_Name** | **Store\_Name** | | East | Boston | | East | New York | | West | Los Angeles | | West | San Diego |   We want to find out sales by store, and we only want to see stores with sales listed in the report. To do this, we can use the following SQL statement using **INNER JOIN**:  **SELECT A1.Store\_Name STORE, SUM(A2.Sales) SALES  FROM Geography A1  INNER JOIN Store\_Information A2  ON A1.Store\_Name = A2.Store\_Name  GROUP BY A1.Store\_Name;**  Result:   |  |  | | --- | --- | | **STORE** | **SALES** | | **Los Angeles** | **1800** | | **San Diego** | **250** | | **Boston** | **700** |   By using **INNER JOIN**, the result shows 3 stores, even though we are selecting from the ***Geography*** table, which has 4 rows. The row "New York" is not selected because it is not present in the ***Store\_Information***table. |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Outer Join**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Previously, we had looked at left join, or inner join, where we select rows common to the participating tables to a join. What about the cases where we are interested in selecting elements in a table regardless of whether they are present in the second table? We will now need to use the **SQL OUTER JOIN** command.  The syntax for performing an outer join in SQL is database-dependent. For example, in Oracle, we will place an "(+)" in the [**WHERE**](http://www.1keydata.com/sql/sqlwhere.html) clause on the other side of the table for which we want to include all the rows.  Let's assume that we have the following two tables,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   Table ***Geography***   |  |  | | --- | --- | | **Region\_Name** | **Store\_Name** | | East | Boston | | East | New York | | West | Los Angeles | | West | San Diego |   and we want to find out the sales amount for all of the stores. If we do a regular join, we will not be able to get what we want because we will have missed "New York," since it does not appear in the ***Store\_Information***table. Therefore, we need to perform an outer join on the two tables above:  **SELECT A1.Store\_Name, SUM(A2.Sales) SALES FROM Geography A1, Store\_Information A2 WHERE A1.Store\_Name = A2.Store\_Name (+) GROUP BY A1.Store\_Name;**  Note that in this case, we are using the Oracle syntax for outer join.  Result:   |  |  | | --- | --- | | **Store\_Name** | **SALES** | | **Boston** | **700** | | **New York** |  | | **Los Angeles** | **1800** | | **San Diego** | **250** |   Note: NULL is returned when there is no match on the second table. In this case, "New York" does not appear in the table ***Store\_Information***, thus its corresponding "SALES" column is **NULL**. |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Left Outer Join**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| In an left outer join, all rows from the first table mentioned in the SQL query is selected, regardless whether there is a matching row on the second table mentioned in the SQL query. Let's assume that we have the following two tables,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   Table ***Geography***   |  |  | | --- | --- | | **Region\_Name** | **Store\_Name** | | East | Boston | | East | New York | | West | Los Angeles | | West | San Diego |   We want to find out sales by store, and we want to see the results for all stores regardless whether there is a sale in the ***Store\_Information*** table. To do this, we can use the following SQL statement using **LEFT OUTER JOIN**:  **SELECT A1.Store\_Name STORE, SUM(A2.Sales) SALES  FROM Geography A1  LEFT OUTER JOIN Store\_Information A2  ON A1.Store\_Name = A2.Store\_Name  GROUP BY A1.Store\_Name;**  ***Result:***   |  |  | | --- | --- | | **STORE** | **SALES** | | **Los Angeles** | **1800** | | **San Diego** | **250** | | **New York** | **NULL** | | **Boston** | **700** |   By using **LEFT OUTER JOIN**, all four rows in the ***Geography*** table is listed. Since there is no match for "New York" in the ***Store\_Information*** table, the Sales total for "New York" is NULL. Note that it is NULL and not 0, as NULL indicates there is no match. |

[**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Cross Join**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A cross join (also called a Cartesian join) is a join of tables without specifying the join condition. In this scenario, the query would return all possible combination of the tables in the SQL query. To see this in action, let's use the following example:  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   Table ***Geography***   |  |  | | --- | --- | | **Region\_Name** | **Store\_Name** | | East | Boston | | East | New York | | West | Los Angeles | | West | San Diego |   The following SQL statement is a Cartesian join between the ***Store\_Information*** and the ***Geography*** tables:  **SELECT A1.Store\_Name STORE1, A2.Store\_Name STORE2, A2.Sales SALES  FROM Geography A1  JOIN Store\_Information A2;**  Result:   |  |  |  | | --- | --- | --- | | **STORE1** | **STORE2** | **SALES** | | **Boston** | **Los Angeles** | **1500** | |  |  |  | | **New York** | **Los Angeles** | **1500** | |  |  |  | | **Los Angeles** | **Los Angeles** | **1500** | |  |  |  | | **San Diego** | **Los Angeles** | **1500** | |  |  |  | | **Boston** | **San Diego** | **250** | |  |  |  | | **New York** | **San Diego** | **250** | |  |  |  | | **Los Angeles** | **San Diego** | **250** | |  |  |  | | **San Diego** | **San Diego** | **250** | |  |  |  | | **Boston** | **Los Angeles** | **300** | |  |  |  | | **New York** | **Los Angeles** | **300** | |  |  |  | | **Los Angeles** | **Los Angeles** | **300** | |  |  |  | | **San Diego** | **Los Angeles** | **300** | |  |  |  | | **Boston** | **Boston** | **700** | |  |  |  | | **New York** | **Boston** | **700** | |  |  |  | | **Los Angeles** | **Boston** | **700** | |  |  |  | | **San Diego** | **Boston** | **700** | |  |  |  |   An alternative way of specifying a cross join is,  **SELECT A1.store\_name STORE1, A2.store\_name STORE2, A2.Sales SALES  FROM Geography A1, Store\_Information A2;**  A cross join is seldom the desired result. Rather, it is an indication that some required join condition is missing in the SQL query. |

  [**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Order By**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| So far, we have seen how to get data out of a table using [**SELECT**](http://www.1keydata.com/sql/sqlselect.html) and [**WHERE**](http://www.1keydata.com/sql/sqlwhere.html) commands. Often, however, we need to list the output in a particular order. This could be in ascending order, in descending order, or could be based on either numerical value or text value. In such cases, we can use the **ORDER BY** keyword to achieve our goal.  The syntax for an **ORDER BY** statement is as follows:  **SELECT "column\_name" FROM "table\_name" [WHERE "condition"] ORDER BY "column\_name" [ASC, DESC];**  The [ ] means that the **WHERE** statement is optional. However, if a **WHERE** clause exists, it comes before the**ORDER BY** clause. **ASC** means that the results will be shown in ascending order, and **DESC** means that the results will be shown in descending order. If neither is specified, the default is **ASC**.  It is possible to order by more than one column. In this case, the **ORDER BY** clause above becomes  **ORDER BY "column\_name1" [ASC, DESC], "column\_name2" [ASC, DESC]**  Assuming that we choose ascending order for both columns, the output will be ordered in ascending order according to column 1. If there is a tie for the value of column 1, we then sort in ascending order by column 2.  For example, we may wish to list the contents of Table ***Store\_Information*** by Sales, in descending order:  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | San Francisco | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   we key in,  **SELECT Store\_Name, Sales, Txn\_Date FROM Store\_Information ORDER BY Sales DESC;**  Result:   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | **Los Angeles** | **1500** | **Jan-05-1999** | | **Boston** | **700** | **Jan-08-1999** | | **San Francisco** | **300** | **Jan-08-1999** | | **San Diego** | **250** | **Jan-07-1999** |   In addition to column name, we may also use column position (based on the SQL query) to indicate which column we want to apply the **ORDER BY** clause. The first column is 1, second column is 2, and so on. In the above example, we will achieve the same results by the following command:  **SELECT Store\_Name, Sales, Txn\_Date FROM Store\_Information ORDER BY 2 DESC;**  The column(s) we use to sort the result do not need to be in the [**SELECT**](http://www.1keydata.com/sql/sqlselect.html) clause. For example, the following SQL,  **SELECT Store\_Name FROM Store\_Information ORDER BY Sales DESC;**  works fine and will give the following result:   |  | | --- | | **Store\_Name** | | **Los Angeles** | | **Boston** | | **San Francisco** | | **San Diego** |   It is also possible to sort the result by an expression. For example, in the following table,  Table ***Product\_Sales***   |  |  |  | | --- | --- | --- | | **Product\_ID** | **Price** | **Units** | | 1 | 10 | 9 | | 2 | 15 | 4 | | 3 | 25 | 3 |   we can use the following SQL to order the results by Revenue (defined as Price \* Units):  **SELECT Product\_ID, Price\*Units Revenue FROM Product\_Sales ORDER BY Price\*Units DESC;**  Result:   |  |  | | --- | --- | | **Product\_ID** | **Revenue** | | **1** | **90** | | **3** | **75** | | **2** | **60** | |

  [**SQL**](http://www.1keydata.com/sql/sql.html) > [**SQL Commands**](http://www.1keydata.com/sql/sql-commands.html) > **Having**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Another thing people may want to do is to limit the output based on the corresponding sum (or any other aggregate functions). For example, we might want to see only the stores with sales over $1,500. Instead of using the [**WHERE**](http://www.1keydata.com/sql/sqlwhere.html) clause in the SQL statement, though, we need to use the **HAVING** clause, which is reserved for aggregate functions. The **HAVING** clause is typically placed near the end of the SQL statement, and a SQL statement with the **HAVING** clause may or may not include the [**GROUP BY**](http://www.1keydata.com/sql/sqlgroupby.html) clause. The syntax for **HAVING** is,  **SELECT ["column\_name1"], Function("column\_name2") FROM "table\_name" [GROUP BY "column\_name1"] HAVING (arithmetic function condition);**  Note: We may select zero, one, or more columns in addition to the aggregate function. If we select zero column, there is no need for the **GROUP BY** clause.  In our example, table ***Store\_Information***,  Table ***Store\_Information***   |  |  |  | | --- | --- | --- | | **Store\_Name** | **Sales** | **Txn\_Date** | | Los Angeles | 1500 | Jan-05-1999 | | San Diego | 250 | Jan-07-1999 | | Los Angeles | 300 | Jan-08-1999 | | Boston | 700 | Jan-08-1999 |   we would type,  **SELECT Store\_Name, SUM(Sales) FROM Store\_Information GROUP BY Store\_Name HAVING SUM(Sales) > 1500;**  Result:   |  |  | | --- | --- | | **Store\_Name** | **SUM(Sales)** | | **Los Angeles** | **1800** | |