

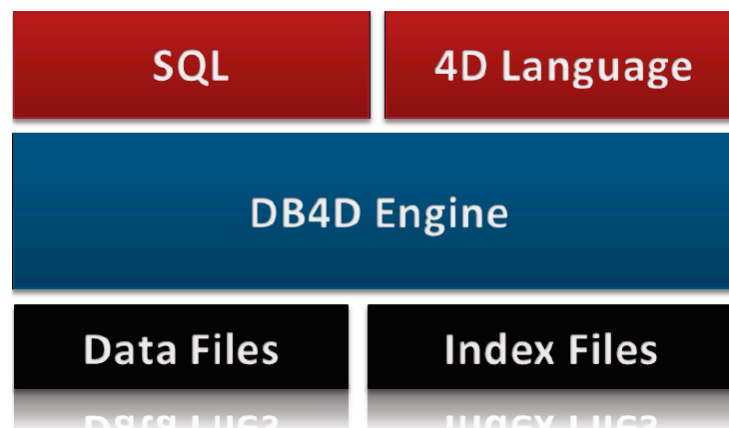
# Beginning SQL

Presented by: **Christopher Visaya**

## AN INTRODUCTION TO SQL

If you have never written a line of SQL, the first thing to do is cast away your trepidation: It's a simple enough language to get a grip on the basics. SQL stands for Structured Query Language and is typically pronounced "S.Q.L." or "sequel". It is a language that is used by relational database management systems to access databases, and create, modify and destroy database objects.

Since the initial release of version 11, 4D developers have had the opportunity to use SQL in 4D. But why would they want to? The simplest answer boils down to it's spread: It is accepted as an industry standard language in RDMS. But don't fret; SQL's entry into 4D does not spoil 4D as you know and love it. Rather, it gives you the flexibility of choice. As the diagram below demonstrates, a query in SQL has the same-level access to the database engine as does a query in the 4D language; neither takes precedence over the other:



This grants the developer the use of the advantages of the 4D query engine in synergy with the advantages of using the SQL query engine for more flexibility to access and manipulate their databases than ever before. So without further adieu, the rest of this document and session4D welcomes you to the world of SQL. We're happy you're choosing our platform to learn the language!

## SQL IN 4D

You can write SQL code in any 4D method; it is a very seamless transition to swap between writing 4D and SQL statements. All you have to do is let 4D know that the next line, or lines, of code is in the SQL language. It is very much akin to starting and ending a For loop, for example. Sometimes it's even easier than that. Here are the three ways we invoke SQL code in 4D:

### QUERY BY SQL

This command allows for simple SELECT statements while still retaining the 4D Current Selection mechanism. In any method, you could implement the following code to look for all Customers from California. Don't worry about the details of the code now – it will all be explained over the course of this document. For now, just be aware that calling QUERY BY SQL tells 4D that a SQL statement sits inside of the following parentheses.

```
QUERY BY SQL([Customers];"State='CA'")
```

```
`The "regular" SQL equivalent to the above is:
`SELECT * FROM Customers WHERE State = 'CA'
`
```

`You can also set the query destination. For example:

```
SET QUERY DESTINATION(Into set ;"set1")
QUERY BY SQL([Customers] ;"State='CA'")
```

### Begin SQL/End SQL Tags

Full SQL statements can be used in the method editor by enclosing all lines within Begin SQL/End SQL tags. You can also run multiple lines as long as each separate statement is separated by a semicolon (;). Note that there is no Current Selection created if running a query this way. Data may be stored in variables, fields, arrays and tables. Unless otherwise stated below, all of the examples in this document will store data into a 4D list box. Wherever you choose to store your query results, you will be able to access it using 4D at that point.

```
Begin SQL
    SELECT Name FROM Customers INTO <<Column1>>;
    SELECT State FROM Customers INTO :Column2;
End SQL
```

### Execute Immediate

Execute Immediate happens inside Begin SQL/End SQL tags, but there is a huge benefit to querying this way: You can build your SQL statements programmatically via 4D. In the example below, the table and destination are both based on the variables \$table and \$destination. You can come up with any algorithm prior to executing this snippet that will decide what these variables are set to. Then, when you are ready, your SQL code will run using the desired values.

```
$table:=Field name(->[Customers]Name)
$destination:="<<Box1>>"
$sql_qry:="SELECT "+$table+" FROM Customers INTO "+$destination
Begin SQL
    EXECUTE IMMEDIATE :$sql_qry
End SQL
```

## SQL JARGON AND BASICS

Let's consolidate some verbiage that will be used henceforth:

### Rows

A row contains data for a single record which includes all fields in the table. In 4D, a record is equivalent to a SQL row.

### Columns

A column contains data for a specific characteristic of the records. In 4D, a field is equivalent to a SQL column.

### Tables

A table is a collection of rows and columns. This is essentially as non-different from what we have been calling tables in 4D.

Figure 1.1 shows an example of columns and rows in a table.

ID	Name	State
C001	Jonathan	CA
C002	Curtis	TX
C003	Christiansen	CA
C004	Bosco	CA
C005	Walter	NV
C006	Nancy	MA
C007	Sue	IL
C008	Lorie	ME
C009	Nicole	DE
C010	Jackie	WY

One Column/Field

One Row/Record

Figure 1.1: Illustration of table, column and row.

### Comments

Use `/*` and `*/` to comment out SQL code within Begin SQL/End SQL tags. You can also comment out the whole block by using the same comment-signifier in 4D, the back quote (```).

### Whitespace

Whitespace is ignored in SQL statements. The following two SQL statements are identical.

```
SELECT * FROM Customers;

SELECT *
FROM Customers;
```

### Case-Sensitivity

SQL is not case sensitive. It is common practice to write reserved words in all-caps.

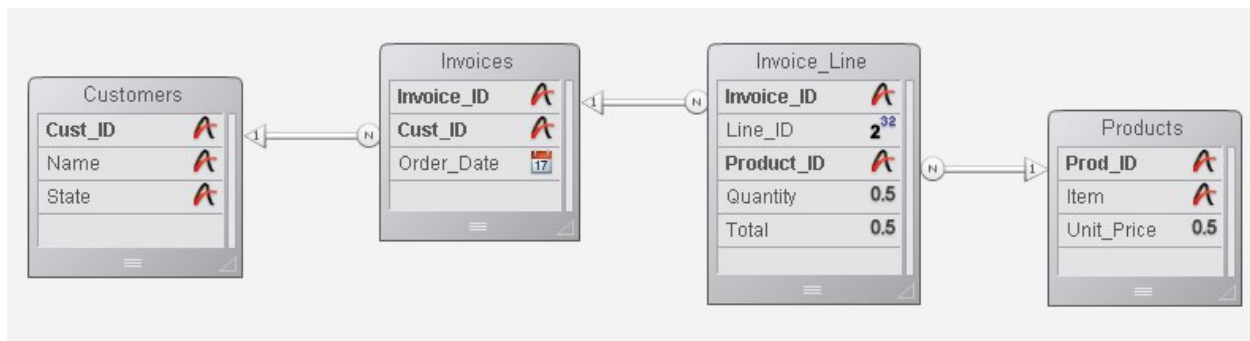


Figure 1.2: Basic Customer-Invoice database.

## THE SELECT STATEMENT

Now let's get started using SQL statements! Figure 1.2 shows the structure used in the examples in this document. The first thing we will take a look at is how to access data from these tables, and the `SELECT` command is used to accomplish this. We can use the `SELECT` statement to make simple requests for a single column or row from a table, or we can use `SELECT` to build up a very complex query. When using the `SELECT` statement, we are doing the SQL equivalent of using a variation of the `QUERY` command in 4D. Keep in mind that *equivalent* is not *identical*; the algorithm behind `SELECT` differs from `QUERY`, but those details are beyond the scope of this session.

Cust_ID	Name	State

**Figure 1.3: The Customers table.**

### Select Everything

Figure 1.3 displays a very basic Customers table in the good old fashioned 4D visual representation that we know. It is comprised of 3 columns (3 fields): Cust\_ID, Name and State. You can't see it now, but there are 20 rows (20 records) in this table, each representing a different customer. In 4D, if we wanted to select all of the records and display them, this would be our method:

```
`Display all Customer records
`
ALL RECORDS([Customers])
DISPLAY SELECTION([Customers])
```

Not much of a query, but Figure 1.4 shows our result in a default 4D output form.

Cust_ID :	Name :	State
C00001	Albert	CA
C00002	Brendan	CA
C00003	Charlie	CA
C00004	Doris	CA
C00005	Eric	CA
C00006	Frank	CA
C00007	Gerald	CA
C00008	Henry	CA
C00009	Iris	KY
C00010	Jean	NY
C00011	Laurent	CA
C00012	Mike	NH
C00013	Nausica	RI
C00014	Oscar	NY
C00015	Patrick	TX
C00016	Quentin	TN
C00017	Romeo	CA
C00018	Sylvie	NV
C00019	Tom	CA
C00020	Ulrich	AR

**Figure 1.4: Using 4D to display all Customer records.**

The SQL statement to display all records is just about as complex as the 4D statement (as in, it's not complex at all), but you will have to become accustomed to the fact that there is no default output form that is called as is with `DISPLAY SELECTION`. In the rest of these notes, we will use the `Begin SQL/End SQL` tags to write our SQL code and we will put all results into a list box, `Box1`, unless otherwise noted. Below is the SQL version to display all records.

```
`Display all customer records in listbox Box1.
`
Begin SQL
    SELECT * FROM Customers INTO <<Box1>>
End SQL
```

#### Code Breakdown

**Begin SQL, End SQL** – Everything in between these two lines will be evaluated as SQL code. You will not be able to use any 4D language statements inside these tags, but you can use 4D arithmetic expressions as input parameters.

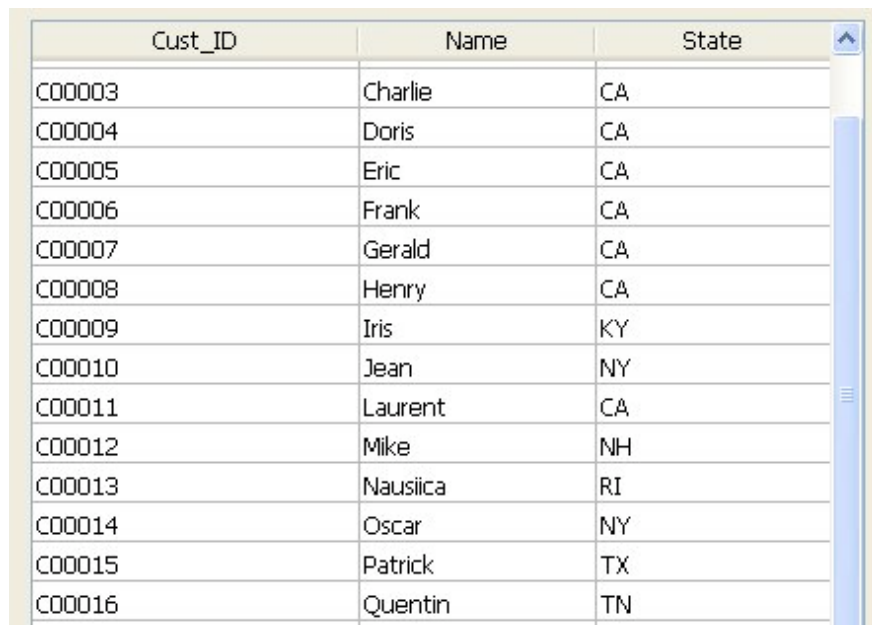
**SELECT** – This tells the SQL engine that we will be retrieving data.

**\*** – The star/asterisk is synonymous with “all”. In this case, select all columns from the table.

**FROM** – This keyword denotes which table to run the query on. Our example uses the Customers table.

**INTO** – In 4D we need to store our SQL queries somewhere to display them. In this case, store it in the list box object, `Box1`. Remember, this is required because there is no default output when you run SQL statements. Note the `<<>>` surrounding the object's variable name. These are necessary when passing variable values between 4D code and SQL code. You can also precede the variable name with a colon (i.e. `:Box1`), but in the interest of being more visually obvious, the prior style was chosen for this document. If you were to run the code without **INTO <<Box1>>** or **INTO :Box1**, the query would still work, but we would have no visual feedback that it did!

Figure 1.5 displays the results for running the SQL code. The results are on par with the 4D results. Note that all records are not displayed.

A screenshot of a 4D list box displaying the results of an SQL query. The list box has a title bar and a scroll bar on the right. The data is presented in a table with three columns: Cust\_ID, Name, and State. The table contains 16 rows of customer data. The list box is titled 'Display all customer records in listbox Box1.'.

Cust_ID	Name	State
C00003	Charlie	CA
C00004	Doris	CA
C00005	Eric	CA
C00006	Frank	CA
C00007	Gerald	CA
C00008	Henry	CA
C00009	Iris	KY
C00010	Jean	NY
C00011	Laurent	CA
C00012	Mike	NH
C00013	Nausiica	RI
C00014	Oscar	NY
C00015	Patrick	TX
C00016	Quentin	TN

**Figure 1.5: Using SQL to display all Customer records.**

### **SELECT specific columns**

One advantage of using SQL is the ability to limit the number of fields, or columns, to include and display in your query. For example, let's say we don't care about the Cust\_ID or State of the customer, we just want their names. In 4D, your code would look identical to above, but your output form would only include the fields that you are interested in. Using SQL, we can control this in the query itself as follows:

```
` Display the names of Customers into the listbox, Box1.
`
Begin SQL
    SELECT Name FROM Customers INTO <<Box1>>
End SQL
```

### **Code Breakdown**

The only difference in this and the previous example is that we took out the star/asterisk (\*). Here, we are naming a specific column, Name, to display. The other two columns in the table will not be displayed. Note that there were no changes to the form itself. The 4D list box is handy enough to add columns accordingly based on the size of the table. This is a neat little feature that is sure to come in handy if you will be outputting potentially drastically different query results into the same form. The results are printed in Figure 1.6.



NAME
Albert
Brendan
Charlie
Doris
Eric
Frank
Gerald
Henry
Iris
Jean
Laurent
Mike
Nausica
Oscar

**Figure 1.6: Using SQL to show only the Name column of the Customers table.**

### **Using ORDER BY to sort**

There will usually be a need to sort query results based on specific situations. For example, a client may want to quickly browse through a search result and wants the list to be alphabetized. Sorting records is done in 4D by using the ORDER BY command. The Customers table in the example is already sorted, so we are going to use another table containing products whose structure is displayed in Figure 1.7. First, we will use 4D to sort.

Prod_ID	Item	Unit_Price
		0.5

Figure 1.7: The Products table.

```
\Sort all of the products alphabetically by name.
\
ALL RECORDS([Products])
ORDER BY ([Products];[Products]Item)
DISPLAY SELECTION([PRODUCTS])
```

These results are displayed in Figure 1.8 based on a default output form:

Prod_ID :	Item :	Unit price :
FOOD004	Beef - stirloin	15.99
FOOD001	Coffee Beans	9.99
FOOD003	Cola	1.59
FOOD004	Olive oil	3.99
MISC003	Plates	1.59
MISC002	Soap	3.99
MISC001	Tulips	7.59
FOOD002	Whole bread	1.99

Figure 1.8: Using 4D to sort products alphabetically.

The SQL equivalent is nearly identical, with results that are equivalent to the above:

```
\
\Query all products and sort them alphabetically. Output to listbox,
\Box1.
\
Begin SQL
    SELECT*FROM Products
    ORDER BY Item INTO <<Box1>>;
End SQL
```

#### Code Breakdown

**ORDER BY** – This tells SQL to sort the results based on the following argument. In our case, it will sort by the item name. We could have picked any other column to sort by if we wanted.

It is also possible to sort by multiple columns, so let's try that. Going back to our Customers table, let's sort our customers by their State, then by their Names. But let's get fancy and sort their names in reverse alphabetical order. Here is the code:

```
\
```

```
`Query all customers and order them by the state they are from  
`first, then order them by their name second in reverse.
```

```
`4D code:
```

```
ALL RECORDS([Customers])  
ORDER BY([Customers];[Customers]State;[Customers]Name;<)  
DISPLAY SELECTION([Customers])
```

```
`SQL code:
```

```
Begin SQL  
    SELECT Name, State FROM Customers  
    ORDER BY State, Name DESC INTO <<Box1>>  
End SQL
```

The ORDER BY command is followed by the first sorting parameter followed by the second. As we can see from the results below, the State column is sorted first, and then each Customer from California is sorted reverse-alphabetically inside. The DESC keyword that follows Name accomplishes this. Figure 1.9 shows the results of the SQL variation of the code. From this point onward, the displayed results will always be from the SQL code unless otherwise noted.

NAME	STATE
Ulrich	AR
Tom	CA
Romeo	CA
Laurent	CA
Henry	CA
Gerald	CA
Frank	CA
Eric	CA
Doris	CA
Charlie	CA
Brendan	CA
Albert	CA
Iris	KY
Mike	NH
Oscar	NY
Nausiica	NY
Jean	NY
Sylvie	TN

**Figure 1.9: Customers ordered by State first, then Name second.**

### Using WHERE to find specific records

When querying the Customers table, we may only be interested in looking at the records of all customers in a specific state. In the Products table, we might only want to see items that cost more than \$7.00. We need some way to set a condition in the query to achieve this. The WHERE clause is used for this very purpose.



In the case of wanting to find customers from a specific state, such as California, we want our output to look like this:

NAME	STATE
Albert	CA
Brendan	CA
Charlie	CA
Doris	CA
Eric	CA
Frank	CA
Gerald	CA
Henry	CA
Laurent	CA
Romeo	CA
Tom	CA

**Figure 1.10: Customers from California.**

Here is the code to achieve the results in Figure 1.10.

```
`Find only customers from California.
`
`4D code:
QUERY ([Customers] ; [Customers] State="CA")
DISPLAY SELECTION([Customers])

`SQL code:
Begin SQL
    SELECT Name,State FROM Customers
    WHERE State = 'CA' INTO <<Box1>>
End SQL
```

### Code Breakdown

WHERE needs a column and an operator to work. The column in our example is State, and the operator in this example is the equal sign (=).

At last! We have narrowed down a search. In celebration of this occasion, we may reward ourselves by taking a look at the other operators available for use with WHERE:

Operator	Details
=	Equals
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater Than or Equal To
<=	Less Than or Equal To
BETWEEN	Values within a range
IN	Values in a specified subset
LIKE	Values that use wildcards
NOT	Negates another operator

Let's try a different operator by querying only products that cost more than my annual salary, \$7.00.

```

`
`Display products that cost more than $7.00
`

`4D code:
QUERY([Products];[Products]Unit_Price>7)
DISPLAY SELECTION([Products])

`SQL code:
Begin SQL
    SELECT Item, Unit_Price FROM Products
    WHERE Unit_Price > 7 INTO <<Box1>>
End SQL

```

### Code Breakdown

The biggest difference to note is that the value operated on in the first example belongs to a field of type Alpha, and thus needed to be enclosed by single quotes (State = 'CA'). In the second example, the value belongs to a field of type Real, and did not need quotes (Unit\_Price > 7). This is a general rule of thumb: Non-numeric values used in WHERE need to be surrounded by single quotes, numeric values do not. The results for our expensive-item search are on Figure 1.11.

ITEM	UNIT_PRICE
Coffee Beans	9.99
Tulips	7.59
Beef	15.99

Figure 1.11: Query results for all products greater than \$7.00.

### BETWEEN, IN, LIKE and NOT

The last four operators are not as obvious to figure out as the rest, and it is worth exploring them a little more in depth. Here are some quick examples of how to use BETWEEN, IN, LIKE and NOT.

#### BETWEEN

BETWEEN returns values that exist within a range. This range is defined immediately after the operator is called.

```

`
`Display all customers whose names begin with G through M
`

`4D code:
QUERY([Customers];[Customers]Name>"G";*)
QUERY([Customers]; & ;[Customers]Name<"M")
DISPLAY SELECTION([Customers])

`SQL code:
Begin SQL
    SELECT Name, State FROM Customers
    WHERE Name BETWEEN 'G' AND 'M'
    INTO <<Box1>>
End SQL

```

NAME	STATE
Gerald	CA
Henry	CA
Iris	KY
Jean	NY
Laurent	CA

Figure 1.12: Query results for customers whose names begin with G through M.

## IN

IN returns values that are included in a specified list. This list follows immediately after IN is called. The list is comma delimited with each element being surrounded by single quotes. Had the list been numeric, we would have omitted the quotes.

```
`Display only the products that have product IDs of FOOD002 and
`FOOD004.
```

```
`4D:
```

```
QUERY([Products];[Products]Prod_ID="FOOD002";*)
QUERY([Products]; | ;[Products]Prod_ID="FOOD004")
DISPLAY SELECTION([Products])
```

```
`SQL:
```

```
Begin SQL
```

```
SELECT Prod_ID, Item FROM Products
WHERE Prod_ID IN ('FOOD002', 'FOOD004')
INTO <<Box1>>
```

```
End SQL
```

PROD_ID	ITEM
FOOD002	Whole bread
FOOD004	Beef

Figure 1.13: Results when searching for only these 2 items.

## LIKE

LIKE returns all values that are similar to a value specified by the user. This operator is typically used in conjunction with the wildcard symbol, %.

```
`
`Display all products in the Food category.
```

```
`4D:
```

```
QUERY([Products];[Products]Prod_ID="FOOD@")
DISPLAY SELECTION([Products])
```

```
`SQL
```

```
Begin SQL
```

```
SELECT Prod_ID, Item FROM Products
WHERE Prod_ID LIKE 'FOOD%'
INTO <<Box1>>
```

```
End SQL
```

PROD_ID	ITEM
FOOD001	Coffee Beans
FOOD002	Whole bread
FOOD003	Cola
FOOD004	Olive oil
FOOD005	Beef

**Figure 1.14: Results for searching for just food-type products.**

## NOT

The NOT operator returns opposite values from the ones specified in the query. For example, we can look for products that are not in the food category, or we could look for customers whose names do not start in the first half of the alphabet. It is important to note the position of NOT in the WHERE statement.

**`Display all customers who are not from California**

**`4D code:**

```
QUERY([Customers];[Customers]State#"CA")
DISPLAY SELECTION([Customers])
```

**`SQL code: (Note the position of NOT)**

**Begin SQL**

```
SELECT Name, State FROM Customers
WHERE NOT State = 'CA'
INTO <<Box1>>
```

**End SQL**

NAME	STATE
Iris	KY
Jean	NY
Mike	NH
Nausiica	RI
Oscar	NY
Patrick	TX
Quentin	TN
Sylvie	NV
Ulrich	AR

**Figure 1.15: All non-Californian customers.**

## WHERE and ORDER BY together

It is possible, and common, to use WHERE and ORDER BY together. After all, just because we limit our search results does not mean that they will come back in the order we desire. Here how to use them simultaneously. The code is still very simple and straightforward, right?

```
`
`Display all customers who are not from California, and sort them by
`state in reverse alphabetical order.

`4D:
```

```

QUERY ([Customers]; [Customers]State#"CA")
ORDER BY ([Customers]; [Customers]State;<)
DISPLAY SELECTION([Customers])

`SQL:
Begin SQL
    SELECT Name, State FROM Customers
    WHERE NOT State = 'CA'
    ORDER BY State DESC
    INTO <<Box1>>
End SQL

```

#### Code Breakdown

A few comments about the code: ORDER BY must always happen after WHERE. Figure 1.16 displays our results.

NAME	STATE
Patrick	TX
Quentin	TN
Nausiica	RI
Oscar	NY
Jean	NY
Sylvie	NV
Mike	NH
Iris	KY
Ulrich	AR

Figure 1.16: Non-Californian customers sorted by reverse alphabetical order by state.

#### MULTIPLE CONDITIONS USING AND, OR

A realistic situation may call for multiple conditions in a query, such as looking for our oldest customers who come from the east. Just like 4D, we use the logical operators AND and OR. The implementation is a little different as seen below.

```

`Display only food-type products that cost under $7.00 and order them in
`ascending order by cost.

`4D code:
QUERY ([Products]; [Products]Prod_ID="FOOD@";*)
QUERY ([Products]; &; [Products]Unit_Price<7)
ORDER BY ([Products]; [Products]Unit_Price)
DISPLAY SELECTION([Products])

`SQL code:
Begin SQL
    SELECT Prod_ID, Item, Unit_Price FROM Products
    WHERE Prod_ID LIKE 'FOOD%' AND Unit_Price < 7
    ORDER BY Unit_Price
    INTO <<Box1>>
End SQL

```

#### Code Breakdown

The code here is straightforward. In the 2<sup>nd</sup> line of the SQL statement, simply add the logical operator (AND) and then include another condition. Figure 2.1 shows the results from the query.

PROD_ID	ITEM	UNIT_PRICE
FOOD003	Cola	1.59
FOOD002	Whole bread	1.99
FOOD004	Olive oil	3.99

**Figure 2.1: Food-type products under \$7.00**

#### A note about the evaluation order of logical operations

SQL processes AND operations before OR. Simple queries that only have one or two conditions are obviously not affected by this. However, more in-depth queries that have more than two conditions could potentially return unexpected results. Here is an example of how to differentiate between A OR (B AND C) versus (A OR B) AND C.

```

`
`Show all customers named Jean, or customers named Brendan from CA
`A OR (B AND C)

`4D code:
QUERY ([Customers] ; [Customers] State="CA" ; *)
QUERY ([Customers] ; & ; [Customers] Name="Brendan" ; *)
QUERY ([Customers] ; | [Customers] Name="Jean")
DISPLAY SELECTION ([Customers])

`SQL code:
Begin SQL
    SELECT Cust_ID, Name, State FROM Customers
    WHERE Name = 'Jean' OR Name = 'Brendan'
        AND State = 'CA'
    INTO <<Box1>>
End SQL

```

#### Code Breakdown

As you can see above, the SQL code was written as A OR B AND C. Since SQL evaluates the AND condition first, it is run as A OR (B AND C), note that it does not matter that the OR condition came first. Figure 2.2 is what we get.

CUST_ID	NAME	STATE
C00002	Brendan	CA
C00010	Jean	NY

**Figure 2.2: A OR (B AND C)**

In this next example, we force SQL to evaluate the OR operation first by encapsulating the condition inside parenthesis. Notice the simplified approach compared to creating sets using 4D.

```

`
`Show all customers from California who are named Jean or Brendan.
`(A OR B) AND C

`4D code:
QUERY ([Customers] ; [Customers] State="CA")
CREATE SET ([Customers] ; "set1")
QUERY ([Customers] ; [Customers] Name="Jean" ; *)

```

```

QUERY ([Customers]; | ; [Customers]Name="Brendan")
CREATE SET ([Customers]; "set2")
UNION ("set1"; "set2"; "set3")
USE SET ("set3")
DISPLAY SELECTION ([Customers])

```

```

`SQL code:
Begin SQL
    SELECT Cust_ID, Name, State FROM Customers
    WHERE (Name = 'Jean' OR Name = 'Brendan')
        AND State = 'CA'
    INTO <<Box1>>
End SQL

```

#### Code Breakdown

Though the difference in 4D code is significantly different in the previous two examples, the only difference in the SQL code is a pair of parenthesis. Surrounding the OR conditional statement in parenthesis causes SQL to evaluate this operation first, and thus the statement is interpreted as (A OR B) AND C. Our results are printed below in Figure 2.3.

CUST_ID	NAME	STATE
C00002	Brendan	CA

Figure 2.3: (A OR B) AND C.

## OTHER SELECTION TRICKS

It is possible to temporarily create a new column on the fly based on a calculation of current column values. Furthermore, you can name the column title. It will exist in temporary memory and will not permanently create a field in your 4D table.

```

`
`Assume sales tax of 8.25% and display all products' values at that
`cost.
`
`4D code:
`In 4D, you could create an output form that includes a field whose
`values contain the calculated values from other fields.
`
`SQL code:
Begin SQL
    SELECT Prod_ID, Item, Unit_Price, Unit_Price * 1.0825 AS With_Tax
    FROM Products
    INTO <<Box1>>
End SQL

```

#### Code Breakdown

##### New columns to contain calculated values

In addition to selecting Prod\_ID, Item and Unit\_Price, there is another column, Unit\_Price\*1.0825. This column is created in memory to display the cost of an item after the 8.25% tax is applied. You can also use addition (+), subtraction (-) and division (/).

##### Aliases

Since our new column has not been created before, we need to give it a name. As you can see in the Figure 3.1, the last column is called "With\_Tax". The alias was assigned using the AS

keyword immediately following the calculated value. If we did not include AS, the column would be labeled "<expression>".

PROD_ID	ITEM	UNIT_PRICE	With_Tax
FOOD001	Coffee Beans	9.99	10.814175
FOOD002	Whole bread	1.99	2.154175
FOOD003	Cola	1.59	1.721175
FOOD004	Olive oil	3.99	4.319175
MISC001	Tulips	7.59	8.216175
MISC002	Soap	3.99	4.319175
FOOD005	Beef	15.99	17.309175
MISC003	Plates	1.59	1.721175

**Figure 3.1: Our temporary column, With\_Tax, contains a calculated value.**

## AGGREGATE FUNCTIONS

Aggregate functions are used to calculate results from multiple records and fields. Below are descriptions of the most common. Note that they will only be applied to fields that contain non-null values. Also be aware that running these functions on tables with hundreds of thousands of records gives results about 3 seconds slower than running the 4D equivalent. On smaller databases, the difference in speed is not noticeable.

Function	Description
COUNT()	Returns the number of records in the field specified.
SUM()	Returns the sum of values in the field specified.
AVG()	Returns the average value of data in the field specified.
MAX()	Returns the largest out of all values in the field specified.
MIN()	Returns the smallest out of all values in the field specified.

### COUNT()

Here is an example of how to use COUNT to find the number of all records in the Customers table.

```
`Count the number of customers`

`4D code:
ALL RECORDS([Customers])
$var:=Records in selection([Customers])
ALERT("There are "+String($var)+" customers")

`SQL code:
Begin SQL
    SELECT COUNT(*) AS TotalCustomers FROM Customers
    INTO <<Box1>>
End SQL
```

#### Code Breakdown

The above code counts all records from the Customers table, and will return 20, verifying the statement made early in these notes that we have that many customers.

### SUM()

It's easy to mix up COUNT and SUM in practice. Just remember that COUNT returns a number of records, while SUM returns a number related to values in the column. In this example, we don't pass \* to look for all records. Instead, we limit our operation to a single column, Quantity, and total up the number of units sold.

```
`Display the total quantity of product FOOD001 sold.`
```



```

`4D:
$sum:=0
QUERY([Invoice_Line];[Invoice_Line]Product_ID="FOOD001")
$sum:=Sum([Invoice_Line]Quantity)
ALERT("There have been "+String($sum)+" FOOD001 items sold.")

`SQL:
Begin SQL
    SELECT SUM(Quantity) AS All_Units FROM Invoice_Line
    WHERE Product_ID = 'FOOD001'
    INTO <<Box1>>
end sql

```

#### Code Breakdown

The above code adds up all of the orders for FOOD001 from the Invoice\_Line table and returns 64. The other operators work the same way as COUNT and SUM, and making them work are left to the bright reader.

## GROUP BY AND HAVING

One of the last things we will look at for querying data is using GROUP BY and HAVING to further refine our search results.

#### Using GROUP BY to sort

GROUP BY causes aggregate functions to be grouped based on columns. Here is an example to show this:

```

`Show the number of orders for each product
`
Begin SQL
    SELECT Product_ID, SUM(Quantity) AS All_Units
    FROM Invoice_Line
    GROUP BY Product_ID
    INTO <<Box1>>
End SQL

```

#### Code Analysis

This SELECT statement uses GROUP BY to show us the total Quantity of each product. Figure 4.1 shows our results.

PRODUCT_ID	All_Units
FOOD001	64
FOOD002	71
FOOD003	90
FOOD004	149
MISC001	78
MISC002	141
MISC003	50

**Figure 4.1: Results grouped by Product\_ID.**

#### Using HAVING to filter data

HAVING is used to filter grouped data. It needs to follow GROUP BY in order to do anything. In the example below, all we are trying to do is to display only states that have more than 1 customer.

```

`Find the number of customers in each state and display only those
`states with more than 1 customer.

```

```

`4D code: Note this will output to a listbox with the correct number of
`columns.

ARRAY STRING(2;$state;0)
ARRAY STRING(2;$newstate;0)
ARRAY INTEGER($count;0)
ARRAY INTEGER($newcount;0)

`Store individual state names into an array, $state.
ALL RECORDS([Customers])
DISTINCT VALUES([Customers]State;$state)
$var:=Size of array($state)

`For each state, count the number of customers.
`If more than 1 customer in the state, copy count and state
`to new arrays, $newcount & $newstate.
For ($i;1;$var)
    QUERY([Customers];[Customers]State=$state{$i})
    INSERT IN ARRAY($count;$i;1)
    $count{$i}:=Records in selection([Customers])
    If ($count{$i}>1)
        APPEND TO ARRAY($newcount;$count{$i})
        APPEND TO ARRAY($newstate;$state{$i})
    End if
End for

`Put the results into columns in a listbox.
COPY ARRAY($newstate;Column1)
COPY ARRAY($newcount;Column2)

`SQL code:
`Begin SQL
    SELECT State, COUNT(Cust_ID) As Customers FROM Customers
    GROUP BY State
    HAVING COUNT(Cust_ID)>1
    INTO <<Box1>>
End SQL

```

### Code Breakdown

There is a big difference between the 4D code and SQL code this time! The simplicity of writing SQL in this case is obvious – we select the columns from the Customers table, group them by state, and only display the ones that have more than one customer into the list box as seen in Figure 4.2.

STATE	Customers
CA	11
NY	3
TN	3

Figure 4.2: States that have more than 1 customer.

## SUBQUERIES

Sometimes we need to search for records based on criteria from a related table. The way we do this in SQL is with subqueries. Subqueries are queries embedded in queries. In our database structure, we have a number of relationships that link our 4 tables together. Refer back to Figure 1.2 to refresh your memory. The concept is the same in SQL in that we can use these relationships to set the criteria on one table for the search in another table.

When would we need to run a subquery? Consider this scenario. We are interested in finding records from the Invoice\_Line table that contain FOOD001, the product code for coffee. The first example shows how ridiculously easy this is in both 4D and SQL.

```
`Find all invoice lines that are for product code FOOD001`
`
`4D Code:
QUERY([Invoice_Line];[Invoice_Line]Product_ID="FOOD001")
DISPLAY SELECTION([Invoice_Line])

`SQL code:
Begin SQL
    SELECT Invoice_ID, Product_ID FROM Invoice_Line
    WHERE Product_ID = 'FOOD001'
    INTO <<Box1>>
End SQL
```

But this brings up speculative point: What good is this data doing for us? The answer is subjective based on the context of the situation, of course. But upon further consideration, it may more useful to ask to find all invoices (instead of just lines on all invoices) that contain orders for FOOD001.

```
`Find all invoices that include orders for product code FOOD001`
`
`4D code:
QUERY([Invoice_Line];[Invoice_Line]Product_ID="FOOD001")
RELATE ONE SELECTION([Invoice_Line];[Invoices])
DISPLAY SELECTION([Invoices])

`SQL code:
Begin SQL
    SELECT Invoice_ID FROM Invoices
    WHERE Invoice_ID IN (SELECT Invoice_ID
        FROM Invoice_Line
        WHERE Product_ID = 'FOOD001')
    INTO <<Box1>>
End SQL
```

### Code Breakdown

In 4D, we used RELATE ONE SELECTION to use the query results from Invoice\_Line to limit our selection in Invoices. This was not a huge leap beyond the first example in this section.

The SQL example shows off a subquery. First we SELECT columns to display from the Invoices table. Our goal is to limit the results to only show invoices that contain FOOD001 invoice lines. So in the WHERE clause, since we cannot set that condition in the Invoices table, we start a new SELECT in the Invoice\_Line table and thus start the subquery. The English way of interpreting the code is “We are querying the Invoices table to look for Invoice\_IDs that exist in the Invoice\_Line table with a value of ‘FOOD001’”. If you look at what’s inside the parentheses, it’s just a simple select statement reprinted here for clarity:

```
SELECT Invoice_ID FROM Invoice_Line
WHERE Product_ID = 'FOOD001'
```

You could write that statement in your sleep. Just remember a few important notes for using subqueries. The subquery must return a single column. In our case, that was Invoice\_ID. The number of values returned is going to be zero, one, or many. If one is returned, you can check for equality, greater than, not equal to, etc. If many results are returned, you need to check if a value is IN the returned set.

But wait, our results still may not be good enough. What if we wanted to know which customers have orders for coffee beans, but do not know the product code? Here now is an even crazier example to illustrate this:

```
`Find all customers from California or Tennessee who have ordered more than 3
`coffee beans on a single order.
`

`4D code:
`first find the 3 coffee bean orderers
QUERY([Invoice_Line];[Products]Item="Coffee beans";*)
QUERY([Invoice_Line]; & ;[Invoice_Line]Quantity>3)
RELATE ONE SELECTION([Invoice_Line];[Customers])
CREATE SET([Customers];"set1")

`find everyone from CA & TN
QUERY([Customers];[Customers]State="Ca";*)
QUERY([Customers]; | ;[Customers]State="TN")
CREATE SET([Customers];"set2")

`merge the two sets
INTERSECTION("set1";"set2";"subfinal")
USE SET("subfinal")

DISPLAY SELECTION([Customers])

`SQL code:
Begin SQL
    SELECT Name, State, Invoice_ID FROM Customers, Invoices
    WHERE Customers.Cust_ID = Invoices.Cust_ID AND
    Customers.State IN ('CA', 'TN') AND
    Invoices.Invoice_ID IN
    (
        SELECT Invoice_Line.Invoice_ID FROM Invoice_Line
        WHERE Quantity > 3 AND
        Product_ID IN
        (
            SELECT Products.Prod_ID FROM Products
            WHERE Item = 'Coffee Beans'
        )
    )
    INTO <<Box1>>
End SQL
```

### Code Breakdown

A few things happen here that are worth putting under a magnifying glass:

```
SELECT Name, State, Invoice_ID FROM Customers, Invoices
WHERE Customers.Cust_ID = Invoices.Cust_ID AND
```

We are selecting fields from multiple tables, so we need to name those tables in FROM. Each table is separated by a comma. Our WHERE clause does 2 things. First, we identify which fields tie the two tables together. In this case the Cust\_ID field from Customers is linked to the Cust\_ID field from Invoices. This was necessary because SQL does not automatically include related fields in their queries; it has to be flagged manually. These lines are followed by a logical AND with:

```
Customers.State IN ('CA', 'TN') AND
Invoices.Invoice_ID IN
```

This is a straightforward AND, just to query only customers from California and Tennessee. The second line invokes our subquery. We use IN instead of “=” because we expect there to be more than one result returned from the subquery.

```
(SELECT Invoice_Line.Invoice_ID FROM Invoice_Line
WHERE Quantity > 3 AND
Product_ID IN (SELECT Products.Prod_ID
FROM Products
WHERE Item = 'Coffee Beans'))
```

Sitting inside our subquery is another subquery! It is needed in this case because the criteria we are revolving our query around, the product name, is 2 tables away from the table we are running our query on.

Looking at the big picture, the deepest level subquery searches for the product, Coffee Beans. The second level finds where Coffee Beans appear 3 times on an invoice line. The top level query finds the names of customers that have invoices whose contents include a line representing an order of more than 3 coffee beans.

Needless to say, subqueries can get messy. But just like any other language out there: Practice, practice, practice! Or hire someone.

## INSERTING RECORDS

Up until now, we have been putting our entire focus into running queries in our databases. Let’s take a brief look now at modifying actual data. The first command we will look at is the INSERT command. Using this command allows you to create a record with data for all fields into a table that already exists.

```
`
`Create a new customer record with Cust_ID = C00021, Name = Raleigh and
`State = `TN`
`
`4D code:
CREATE RECORD([Customers])
[Customers]Cust_ID:="C00021"
[Customers]Name:="Raleigh"
[Customers]State:="TN"
SAVE RECORD([Customers])

`SQL code:
Begin SQL
    INSERT INTO Customers (Cust_ID, Name, State)
    VALUES ('C00021', 'Raleigh', 'TN')
End SQL
```

### Code Analysis

In the first line of code, INSERT requires that you list the names of columns that you will be inserting. If you will be adding a record complete with values for all columns, it is not necessary to list the names. We did it here for the sake of readability.

The second line of code is where we set the actual values after the VALUE keyword. Be careful when adding records using INSERT. You will need to have a good knowledge of what fields are in your structure as well as their types or your data might not be added properly.

Since we are editing data instead of querying it, there is no output for the code above. If you take a look at your Customer table entries, you will see that the latest entry is indeed Raleigh from Tennessee.

## UPDATING RECORDS

The UPDATE statement lets you update fields for any records in the table. There are many instances where this could come in handy, such as if you needed to do a mass update for similar records. In our case, we're just going to update one record. Bear in mind that this command, like any other command where you modify data, could be potentially damaging. Take extra care to make sure that the condition with which you focus your UPDATE is correct.

```
`Update a customer record by changing Raleigh's name to Chris.
```

```
`4D Code:
QUERY([Customers];[Customers]Name="Raleigh")
[Customers]Name:="Chris"
SAVE RECORD([Customers])
```

```
`SQL Code:
Begin SQL
    UPDATE Customers SET Name = 'Chris'
    WHERE Name = 'Raleigh'
End SQL
```

### Code Breakdown

UPDATE – Signifies that we will be updating an existing record. This statement is followed by SET. As the name of the keyword implies, this is where we set the new value of the data in the column.

The second line of code sets our condition; we will only update the names of records that have the Name field set to Raleigh.

Below is a quick example of how you could potentially ruin your data: Everyone from Arkansas will be graced (or cursed) with my name!

```
`Change the name of all customers from Arkansas to Chris.
```

```
`4D Code:
QUERY([Customers];[Customers]State="AR")
[Customers]Name:="Chris"
save record([Customers])
```

```
`SQL Code:
Begin SQL
    UPDATE Customers SET Name = 'Chris'
    WHERE State = 'AR'
End SQL
```

## DELETING RECORDS

It goes without saying that this command is destructive by nature. Using it will let you delete one or more records in a table. Be Careful!

```
`Delete the entry for Raleigh from the Customers table.
```

```
`4D Code:
QUERY([Customers];[Customers]Name="Raleigh")
DELETE RECORD([Customers])
```

```
`SQL Code:
Begin SQL
    DELETE FROM Customers
    WHERE Name = 'Raleigh'
End SQL
```

### Code Breakdown

Any record that has the value of Raleigh in the Name column will be deleted. As we saw from the previous example, we only have one Raleigh. But if we applied DELETE to the customers in Arkansas... Well, we only have one customer from Arkansas too... But if we had more than one...

## CREATING A TABLE

It is also possible to programmatically create and edit tables. In order to do that, we use the CREATE TABLE command. We give the table a name and include columns and column types. Before we do that, please note that the ability to create tables programmatically has been removed from the 4D language. It used to be included in the 4D Pack expansion, but the latest version of 4D Pack for 4D v11 does not include these commands.

```
`Create a table, Suppliers, that includes fields for the suppliers's  
`names as well as the products they specialize in.
```

```
Begin SQL  
    CREATE TABLE Suppliers  
        (S_Name          VARCHAR(25) ,  
         S_Specialty     VARCHAR(25))  
End SQL
```

### Code Breakdown

The first line of code is straightforward: We are creating a Suppliers table. Everything in the parentheses is fields and their corresponding types. There are other file types out there, including INT, BLOB, TEXT. In Figure 5.1, you can see the new table.

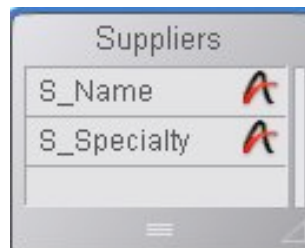


Figure 5.1: Table created in SQL.

## ALTERING A TABLE

The next thing you could do is modify an existing table. This is done with the ALTER TABLE command.

```
`In the Customers table, add a City column of type Alpha.  
`
```

```
Begin SQL  
    ALTER TABLE Customers  
        ADD City VARCHAR(20)  
End SQL
```

Cust_ID	A
Name	A
State	A
City	A

**Figure 5.2: Using SQL to create the City field.**

### Code Breakdown

The Customers table is updated to show that it now has a new field, City. This field can be manipulated just like any other field in 4D and SQL. Figure 5.2 shows our new field in our table. You can start adding data to this field if you like, or you can remove it if it was a mistake:

```
`Remove the City column from the Customers table.

Begin SQL
    ALTER TABLE Customers
    DROP City
End SQL
```

### Code Breakdown

Using the DROP keyword in conjunction with ALTER TABLE, the City column is now removed from the Customers table.

### Deleting a Table

Here is how to delete a table. I won't warn you to be careful.

```
`
`Remove the Suppliers table.
`

Begin SQL
    DROP TABLE Suppliers
End SQL
```

### Code Breakdown

DROP TABLE completely removes the table. If you are certain you want to do this, this command could come in handy.

### NULLS

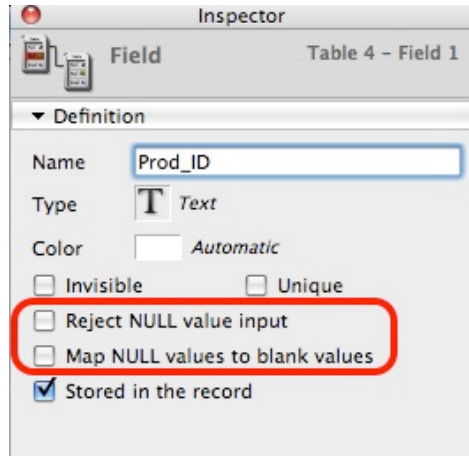
A NULL is a value that exists but is unknown. How does that help the developer when designing their database? Consider the following scenario: You are tracking rates on goods from a small, independent supplier over the year. In October, the building suffers heavy fire damage and the owner shuts down business operations for 2 weeks. At the end of the year, you are interested in seeing the average daily cost per product from the supplier. How do you treat the 2-week period where the business was down?

- Set the values during that time to a default value when calculating the daily average
- Ignore the 2 weeks when calculating the daily average

Depending on what you are trying to accomplish, it may make more sense to ignore the 2-week period. After all, just because the goods did not sell, that does not mean that they had no value. They DID have value, but due to the fire, it was unknown what those values were. In this situation, we would be treating the values during these 2 weeks as NULLs. On the other hand, it may make more sense to default the values to known values such as a zero or the



last-known good value. In this case, we would be treating the values during these 2 weeks as known values, not NULLs. The choice heavily depends on what you are trying to accomplish in your report. The good news is that 4D provides you with the flexibility to use NULLs or not.



With the two options available in a field's properties, developers can choose to reject NULLs altogether, or default them to zeros or blanks.

## CONCLUSION

Of all the little handbook-like documents on the internet for getting started with SQL, you now have a little handbook-like document for getting started with SQL in 4D. Remember that this is far from a comprehensive list of everything that can be done in the language. Implementing SQL into your databases is completely optional, of course, but it never hurts to acquire a new skill.

4D University provides online training for beginning SQL, among other topics including uses with NULLs, accessing system tables and connecting to external databases. Visit [www.4d.com/training](http://www.4d.com/training) for more information.