STEP1® SOFTWARE SOLUTIONS
ONE STEP AHEAD

STEP1 Software Solutions
Advanced Query & View Concepts Session

Relational Databases / Spreadsheets
Views and View Based Query
SQL Queries
- **SQL Structure**
  - **Select, From, Where**

Grouping and Sorting Results
- **Group By**
- **Order By**

Joins
- **ID's, Common fields**

SQL Functions
- **Text Modifiers / Functions**
  - **LEN(), COUNT(), STR(), CAST(), CONVERT(), SUBSTRING(x,x,x)**
- **Aggregate Functions**
  - **SUM, MAX, MIN, TOP, PERCENT**
- **Date Functions**
  - **MONTH(InvoiceDate), Year(InvoiceDate)**

The query module can be used to build quick and easy ad-hoc reports, but to get the most from the module, understanding of the more advanced concepts such as joining views and tables and aggregate functions will give you much more flexibility in building your reports.

Step1 uses a relational database to contain and organize data. A relational database keeps data stored in tables that relate to each other in some way. These tables are organized around a Primary Key, which can then be referenced from another table to link the data together.

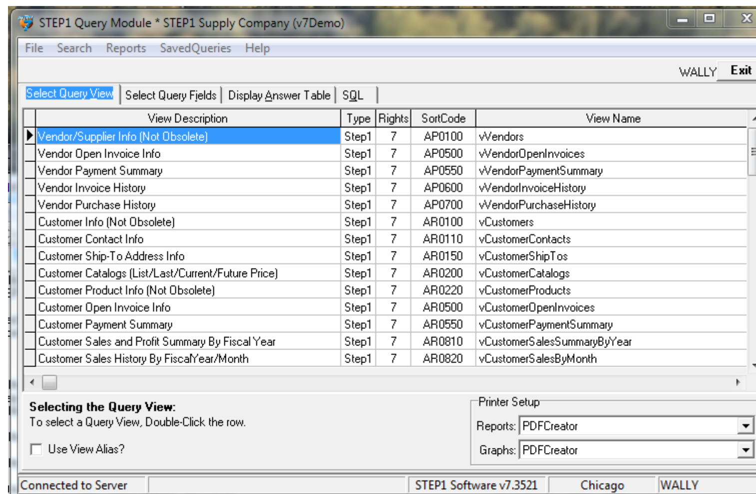
A database consisting of independent and unrelated tables serves little purpose (you may consider to use a spreadsheet instead). The power of relational database lies in the relationship that can be defined between tables. The most crucial aspect in designing a relational database is to identify the relationships among tables. The types of relationship include:

1. one-to-many
2. many-to-many
3. one-to-one

(See Advanced Query Notes at the end of this document)

In this session we will be discussing Data Views and how they can be used for Queries and Reports based on your Step1 Data. This session assumes you know how to use the Step1 Query Module, and at least have an idea what SQL (Structured Query Language) is.

Views and Tables in Query
STEP1's Query module is built to work with views and tables, and allows you to create your own views for additional flexibility. Views in SQL can be thought of as a data set that is built on the fly to further query data from and restrict results to just the data you want to see. Step1 uses a naming convention to make it easy to tell when you are working with a view or table. All Step1 views are named with a small 'v' beginning the name to identify them as views. Additionally, view names should not contain any spaces. (View descriptions can contain spaces.)



1. Data Views vs. Data Tables

- Data Tables hold the actual data. Designed to optimize Transaction Processing.
- A Data View is a specific 'view' into your data.
- Views are sometimes called 'virtual tables'.
- A View is a 'layer' between you and your data that hides the complexity of the data table structure (which was NOT designed for easy human querying and reporting).

2. Advantages of using Views

- Views can be easily modified to adapt to different user's data needs (the Data Tables cannot be easily changed without breaking the programs that access the data).
- If Views are used for queries and reports, then the potential impact of Step1 version updates (and related data table changes) will be minimized. When queries and reports are based on the actual data tables, changes to the data table structures could easily break your queries and reports.
- Primary advantage is that Views can hide the complexity of the SQL required to query data (ie Joins, Aliases, Function Calls, etc), making it much easier to do queries and reports.

Example: Suppose you wanted to query invoice detail for the items for a specified Supplier Acct that were sold to a specified Customer Acct for a specified invoice date range.
If you wrote your query using the actual Step1 data tables, you would have to join 5 tables to get at the data, using the following SQL:

SELECT hed.InvoiceNum, hed.InvoiceDate, itm.ItemCode, det.NumShipped, det.Price
FROM ARInvDet det
Left Outer Join ARInvHed hed On det.ARInvcID = hed.ARInvcID
Left Outer Join ICItems itm On det.ItemID = itm.ItemID
Left Outer Join APVendor sup On det.VendorID = sup.VendorID
Left Outer Join ARCusts cus On hed.CustID = cus.CustID
WHERE hed.InvoiceDate between '2011-1-1' and '2011-12-31'
and cus.CustAcct = 'abbot' and sup.VendorAcct = 'baywest'

But if you wrote your query using the standard Step1 View called vCustomerInvoiceDetail View, NO Joins or Aliases are even needed! Here is the same SQL query using the view:

SELECT InvoiceNum, InvoiceDate, ItemCode, NumShipped, Price
FROM vCustomerInvoiceDetail
WHERE InvoiceDate between '2011-1-1' and '2011-12-31'
and CustAcct = 'abbot' and SupplierAcct = 'Baywest'

3. Using the new View-based Query module in v7.34+

- Standard Step1 views vs. User views
- Step1 View Auto-Updates (via DBQ/DBComm 'Check for Report and View Updates')

- View & Field Level Rights to restrict access to sensitive data
- SQL Tab (User option)
- Use the 'Order By' clause to make sure queries & reports are sorted properly
- Migrating your Saved SQL Queries & Reports to Views (if/when you run into errors)

4. Building Reports using Views

- The hardest part of building reports is usually the initial data issues (setting up a data view in report builder).
- If you use standard Step1 Views (or User views that are properly setup ahead of time using the Query module), report building is pretty basic.
- If necessary, Step1 Support can help you create the User Views for your reports (if standard Step1 views are not sufficient)

SQL Query Structure

SQL Select Queries follow a basic structure that will be used over and over. Select queries are built using three main parts; a Select statement, a From statement, and a Where clause to help define a limited data set.

```
Select *  -- In SQL the asterisk (*) or star is used as a wild card to “Show All Fields”.
From vInventoryItems
Where ObsoleteFlag = ‘Y’
```

This query says *“show all fields for all items from the inventory items view that are marked obsolete.”*
The text following the select statement is a remark added to clarify the coding. To comment a single line use two hyphens. To comment a block or paragraph of text, begin the comment with a slash asterisk (/*) and end with asterisk slash (*/). Here’s an example of a block comment:

```
/*
Block comment. This can be useful to document your queries and makes it easy to describe what the query does so that other users can follow the query
*/
```

Now that we’ve looked at the basic query structure and how to add comments, let’s add some useful code. The previous query is useful, but doesn’t guarantee the result set will be organized in any particular way:

```
Select * From vInventoryItems Where ItemType = ‘I’
```

To make this more useful, we can add a fourth line to specify the ordering of the result set.

```
Select * From vInventoryItems Where ItemType = ‘I’
Order By ItemCode
```

The bottom line of this query will cause the result set to be sorted by ItemCode. In order to sort in reverse order, you can add a qualifier; DESC.

```
Order By ItemCode DESC  -- This will sort by item code in a descending order.
```

Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

Typically, these operators are entered into the Query Value column, after selecting a beginning query view.

Customer Product Info (Not Obsolete)

Select Query View | Select Query Fields | Display Answer Table | SQL

Include	Column	Rights	Field Name	Query Value
☑	1	7	CustAcct	
☑	2	7	CustomerName	LIKE '%CHIC%'
☑	3	7	ItemCode	
☑	5	7	ItemType	
☑	6	7	ItemDescription	
☐		7	ItemExtendedDescription	
☑	7	7	ItemCategory	
☐		7	CategoryDescription	
☑	9	7	ItemSubCat	

```
/* This query shows all customers and all products for that customer, with the current price and last sale date for each
product. We start with the vCustomerProducts view, and restrict it to exclude the Cash account. Note the Where clause
using "NOT LIKE" */
SELECT
 V.CustAcct,
 V.CustomerName,
 V.ItemCode,
 V.CurrentPrice,
 V.LastSaleDate,
 V.SmanCode,
 V.SalesmanName
FROM vCustomerProducts V
WHERE V.CustAcct NOT LIKE '%CASH%'
Order By v.CustAcct, V.ItemCode
```

Different SQL JOINs

Before we continue with examples, let's explore the SQL JOINs you can use:

- INNER JOIN: Returns all rows when there is at least one match in BOTH tables
- LEFT JOIN: Return all rows from the left table, and the matched rows from the right table
- RIGHT JOIN: Return all rows from the right table, and the matched rows from the left table
- FULL JOIN: Return all rows when there is a match in ONE of the tables. (Full joins are seldom used.)

It's even possible to join the same table multiple times, in order to query out specific years for example.

In order to make joins easier, we can alias (or nickname) the views we want to join together. An alias is a way to name a table with a reference, and then use that reference to refer to the table:

```
/* This query shows the highest last sale date each customer has. This example illustrates the alias' for table names, as well as introducing the "Max" function. */
SELECT
 V.CustAcct,
 V.CustomerName,
 MAX(V.LastSaleDate) as LastCustSale
FROM vCustomerProducts V
WHERE V.CustAcct NOT LIKE '%CASH%'
GROUP BY V.CustAcct, V.CustomerName
Order By v.CustAcct

/* Here's an example of a join. We start with the customers view and link in the contacts view, to show the email address for the office contact. Note the alias' and join conditions. */
SELECT
  Cus.CustAcct,
  Cus.CustomerName,
  Cus.LastSaleDate,
  Cus.OfficeContactFirstName,
  Cus.OfficeContactLastName,
  Con.EmailAddress,
  Cus.WebAddress
FROM vCustomers Cus
 Left Join vCustomerContacts Con on Con.CustID = Cus.CustID
  and Con.LastName = Cus.OfficeContactLastName

/* Another example of a join, beginning with the customers view, and joining in the item sales table to show all customers, even those without any item sales. */
SELECT
  cus.CustAcct,
  cus.CustomerName,
  cus.Address1,
  cus.Address2,
  cus.City,
  cus.State,
  cus.Zip,
  cus.SmanCode,
  cus.SalesmanName,
  sales.ItemCode,
  sales.fiscalyear
FROM vCustomers cus
  Left Join vCustomerProductSalesByMonth  Sales
    on sales.custid = cus.custid and sales.smancode = cus.smancode

/* And, finally, an example of linking the customer view with the customer product view, which will show all customers, even those with no customer products. */
SELECT
  V.CustAcct,
  V.CustomerName,
  Prod.ItemCode
FROM vCustomers V
 Left Join vCustomerProducts Prod on Prod.CustID = V.CustID

/* This one is interesting to note the way the join is done will determine whether a customer account number will show in the results. We can look at the query either way. (Show using query on the screen.) This example pulls the customer name from the customer product view, and it's easy to see the blanks. */
SELECT
  Prod.CustAcct,
  V.CustomerName,
  Prod.ItemCode
```

```
FROM vCustomers V
 Left Join vCustomerProducts Prod on Prod.CustID = V.CustID
```

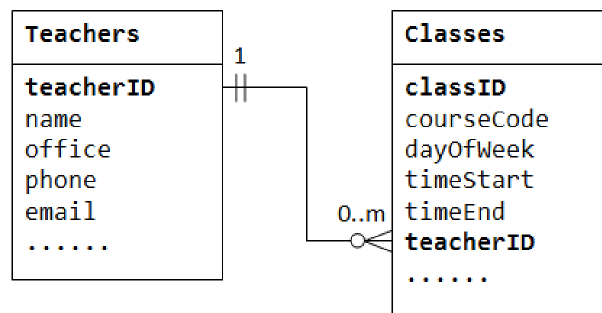
Let's look at some other examples of the built in SQL functions, and a subquery example.

```
/* This query shows all pending orders with an order date of today. */
SELECT
 V.InvoiceNum,
 V.OrderDate,
 V.CustAcct,
 V.TotalDue
FROM VPendingCustomerOrderSummary V
WHERE v.OrderDate = CAST(GETDATE() AS DATE)

/* This query gives the number of items in each sub category. */
SELECT
 V.SubCatCode,
 COUNT(v.ItemCode) as NumItems
FROM vInventoryItems V
GROUP BY V.SubCatCode

/* Sql to trim characters from right side of code */
SELECT ItemCode, Left(ItemCode, LEN(ItemCode) - 3) AS MyTrimmedColumn
From ICItems
Where ItemID > 100
and BreakCaseFlag = 'N'

/* Here's an example of a Case statement using the date function MONTH to select monthly sales balances from the
Customer Sales By Month view. The Case statement checks for validity in each instance, and returns the first data that
matches the criteria, or else a '0' (in this case). */
Select CustAcct, CustomerName, SmanCode,
CurMonthSales =
        Case
                When MONTH(GetDate()) = 1 then SalesBal1
                When MONTH(GetDate()) = 2 then SalesBal2
                -- etc
                When MONTH(GetDate()) = 11 then SalesBal11
                When MONTH(GetDate()) = 12 then SalesBal12
        Else 0
        END
From vCustomerSalesByMonth
Where SmanCode = 'Wally' and FiscalYear = YEAR(GetDate())

/* And, finally, an example of a sub-query. This example shows how you can use a nested query, or sub-query to define
the search criteria. */
Select  CustAcct,   CustomerName,   Address1,  Address2,  City,  State,  Zip
From vCustomers Where CustAcct NOT IN
 (SELECT CustAcct FROM vCustomerInvoiceSummary
   Where InvoiceDate >= '2014-01-01' and TotSales > 0)
```

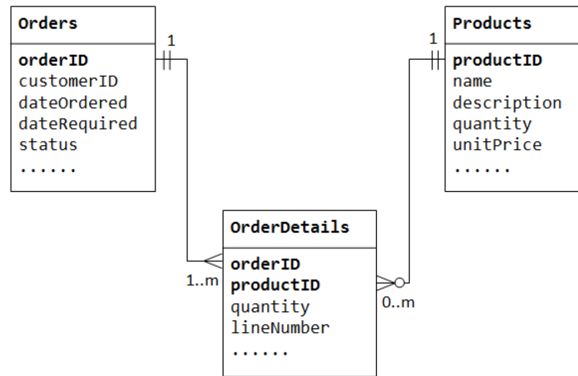
Database Relation Notes

Here are examples of the different relationships:

One-to-Many
In a "class roster" database, a teacher may teach zero or more classes, while a class is taught by one (and only one) teacher. In a "company" database, a manager manages zero or more employees, while an employee is managed by one (and only one) manager. In a "product sales" database, a customer may place many orders; while an order is placed by one particular customer. This kind of relationship is known as *one-to-many*.
One-to-many relationship cannot be represented in a single table. For example, in a "class roster" database, we may begin with a table called Teachers, which stores information about teachers (such as name, office, phone and email). To store the classes taught by each teacher, we could create columns class1, class2, class3, but faces a problem immediately on how many columns to create. On the other hand, if we begin with a table called Classes, which stores information about a class (courseCode, dayOfWeek, timeStart and timeEnd); we could create additional columns to store information about the (one) teacher (such as name, office, phone and email). However, since a teacher may teach many classes, its data would be duplicated in many rows in table Classes.
To support a one-to-many relationship, we need to design two tables: a table Classes to store information about the classes with classID as the primary key; and a table Teachers to store information about teachers with teacherID as the primary key. We can then create the one-to-many relationship by storing the primary key of the table Teacher (i.e., teacherID) (the "one"-end or the *parent table*) in the table classes (the "many"-end or the *child table*), as illustrated below.



Many-to-Many
In a "product sales" database, a customer's order may contain one or more products; and a product can appear in many orders. In a "bookstore" database, a book is written by one or more authors; while an author may write zero or more books. This kind of relationship is known as *many-to-many*.
Let's illustrate with a "product sales" database. We begin with two tables: Products and Orders. The table products contains information about the products (such as name, description and quantityInStock) with productID as its primary key. The table orders contains customer's orders (customerID, dateOrdered, dateRequired and status). Again, we cannot store the items ordered inside the Orders table, as we do not know how many columns to reserve for the items. We also cannot store the order information in the Products table.
To support many-to-many relationship, we need to create a third table (known as a *junction table*), says OrderDetails (or OrderLines), where each row represents an item of a particular order. For the OrderDetails table, the primary key consists of two columns: orderID and productID, that uniquely identify each row. The columns orderID and productID in OrderDetails table are used to reference Orders and Products tables, hence, they are also the foreign keys in the OrderDetails table



One-to-One
In a "product sales" database, a product may have optional supplementary information such as image, moreDescription and comment. Keeping them inside the Products table results in many empty spaces (in those records without these optional data). Furthermore, these large data may degrade the performance of the database.
Instead, we can create another table (says ProductDetails, ProductLines or ProductExtras) to store the optional data. A record will only be created for those products with optional data. The two tables, Products and ProductDetails, exhibit a *one-to-one relationship*. That is, for every row in the parent table, there is at most one row (possibly zero) in the child table. The same column productID should be used as the primary key for both tables.
Some databases limit the number of columns that can be created inside a table. You could use a one-to-one relationship to split the data into two tables. One-to-one relationship is also useful for storing certain sensitive data in a secure table, while the non-sensitive ones in the main table.