Microsoft Dynamics NAV 2013 SQL Readiness Training

Module 2: Query Objects

Microsoft

Version 1.0

Released: October 1, 2012

Conditions and Terms of Use

Microsoft Confidential

This training package content is proprietary and confidential, and is intended only for users described in the training materials. This content and information is provided to you under a Non-Disclosure Agreement and cannot be distributed. Copying or disclosing all or any portion of the content and/or information included in this package is strictly prohibited.

The contents of this package are for informational and training purposes only and are provided "as is" without warranty of any kind, whether express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Training package content, including URL and other Internet Web site references, is subject to change without notice. Because Microsoft must respond to changing market conditions, the content should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Copyright and Trademarks

© 2012 Microsoft Corporation. All rights reserved.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

For more information, see **Use of Microsoft Copyrighted Content** at
http://www.microsoft.com/about/legal/permissions/

Microsoft®, Microsoft Dynamics®, SQL Server®, Internet Explorer®, and Windows® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Microsoft products mentioned herein may be either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are property of their respective owners.

About the Authors	
Author:	Gerard Conroy
Bio:	Gerard is an Escalation Engineer in the Microsoft Dynamics NAV support team based in the United Kingdom. He has been working with Microsoft Dynamics NAV since 2007 and his previous roles within Microsoft include the SQL Server support team and the internal IT Department.
Author:	Christine Avanessians
Bio:	Christine is a Microsoft Dynamics NAV Program Manager working on the Server and Tools team based in Denmark who has provided the content and information for this Module.

Table of Contents

OVERVIEW **1**
LESSON 1: WHEN TO USE THE QUERY OBJECT **2**
Using the Record API 3
ReadState 5
ReadUncommitted 6
ReadShared 6
ReadExclusive 6
Reports 8
XMLports 8
LESSON 2: LINKING SUBTLETIES **10**
LESSON 3: FILTERING SUBTLETIES **16**
LESSON 4: IMPLEMENTING QUERIES FOR ADHOC REPORTING AND BI **20**
OData Enabled Queries 20
TopNumberOfRows and $top 20
OrderBy and $orderby 22
$filter 22
Queries for Charting 23
LESSON 5: QUERY OBJECTS AND PERFORMANCE **27**
FlowFields in Queries 27
Covering Indexes 27
Covering SIFT Indexes 27
Differences Between Query and Record Result Sets 28

Overview

In this module, we will discuss the Query Object, which is a new feature of Microsoft Dynamics NAV 2013.

The Query Object is the newest member of the Object Designer family and the C/AL programming language. This has been added to help you create better and faster solutions. Queries encapsulate a definition of ***what*** data you want out of the system and the corresponding runtime components in the backend engine to ***extract/calculate*** that data. Like its sibling Objects, the Query Object includes a tabular designer for modeling and a corresponding programmatic API to access its features and contents from within C/AL.

There are many different scenarios that the new Query object enables and it provides the opportunity for better performance in many scenarios.

The material below has been designed to complement the level 100/200 training on Query Objects and therefore assumes that students are already familiar with Query basics.

What You Will Learn

After completing this Module, you will understand the capability of the new Microsoft Dynamics NAV Query Object as well as its limitations and some of the more subtle behavior that can be observed when working with complex data structures. Specifically you will learn:

- When to use the Query Object
- The subtleties of linking multiple Data Items in an Query Object
- The subtleties of filtering within and on top of a Query Object
- The subtleties when implementing Query Objects for Ad-Hoc Reporting and BI requirements.

Lesson 1: When to use the Query Object

If you are familiar with SQL, the Query Object is similar to a modeled SQL **SELECT** statement, which can be used in C/AL. If you are new to SQL, the Microsoft Dynamics NAV Query allows you to model the expression that will extract all the data that you need to potentially locate within multiple tables in the database. Does this sound similar to records or maybe even reports? Not quite.

Functionally, there is a long list of capabilities that the query object provides, starting with one of the fundamentals - the ability to ***select*** a subset of fields from multiple tables ***joined*** with different linking criteria (for SQL experts: the "SELECT" and "FROM" clauses). We have taken the first few steps on the path to decoupling the logical representation desired by an end-user from the physical representation in the database. For example, because the data to show a KPI such as sales per customer for each salesperson is spread throughout the database (i.e. in the 'My Customers', and 'Customer' tables), this does not logically mean that the end user would like to see them separately. Digging further, the query provides the ability to ***filter*** and ***order*** the resulting dataset, as well as, return only the ***top*** number of entries (for SQL experts: "WHERE", "HAVING", and "ORDER BY" T-SQL clauses, plus "TOP"). For example, the end user may want to see customers ordered by customer location or by sales amount and filtered by customer number or item code. Finally, it is possible to ***aggregate*** on fields to provide totals ***grouped by*** other fields (for SQL experts: aggregation functions such as "SUM", "AVG", "MIN", "MAX", COUNT and "GROUP BY" clause).

In Microsoft Dynamics NAV 2013, we have discontinued the native database, which means that the product in general and the Query objects in particular, can be highly optimized for the Microsoft SQL Server stack. The Query Object's metadata through a series of translations is eventually expressed as a T-SQL "SELECT" statement (i.e. the syntax to request data from a SQL database) in the Data Access layer of the Microsoft Dynamics NAV Server Tier (NST). All of the functionality provided by the Microsoft Dynamics NAV Query Object is implemented by the corresponding SQL Server TSQL functions (although TSQL provides many more functions that are not implemented in the Query Object). Thus, the NST is able to transfer the work of selecting, joining, filtering, or aggregating over to SQL Server; in short, allowing SQL Server to do what it does best. This provides large performance gains when compared to some of the other Microsoft Dynamics NAV objects, since SQL Server is highly optimized for performing such queries.

We have even added further optimizations to automatically utilize any Covering SIFT indices that may be applicable for queries. A Covering SIFT index is one which includes all the columns which are required on the relevant table to satisfy the Query, i.e. columns which are included in the SELECT, Link or Filter portions of the statement. This gives SQL Server the possibility to satisfy the query from the Covering index without having to read the underlying table, thus saving on expensive I/O operations.

While queries have been optimized for performance, there is still one caveat to all that have just been described. In Microsoft Dynamics NAV 2013, Query result sets are not cached.

© 2012 Microsoft Corporation. All rights reserved.

Depending on the usage pattern, the gains of utilizing a query can be diminished in certain scenarios, especially if the same data is read frequently within a short period of time.

Using the Record API

There are some similarities between Queries and its sibling object in Microsoft Dynamics NAV Records, in that both are a means to access the data stored in the database. However, the Record API only provides a means to get filtered data from a single table; the rest of the operations needed are done in code. While, it may be possible to accomplish the same functional goals (with a small caveat described below), the implementation and results are different. Let us look at the following simple code example.

```
1   //Item is a variable of Type = Record and SubType=Item
2   //ItemLedgerEntry is a variable of Type = Record and SubType=Item Ledger Entry
3   //OutputData is a function that writes out data to a variety of sources
4
5   count := 0;
6   IF Item.FINDSET THEN
7     REPEAT
8       PrevDate := 0D;
9       TotalQty := 0;
10      ItemLedgerEntry.SETCURRENTKEY("Item No.","Posting Date");
11      ItemLedgerEntry.SETRANGE("Item No.",Item."No.");
12      ItemLedgerEntry.SETRANGE("Entry Type", ItemLedgerEntry."Entry Type"::Sale);
13      IF ItemLedgerEntry.FINDSET THEN
14        REPEAT
15          IF (ItemLedgerEntry."Posting Date" <> PrevDate) AND (PrevDate <> 0D) THEN BEGIN
16            OutputData(1, Item."No.",Item.Description,PrevDate,-TotalQty);
17            TotalQty := 0;
18            count := count + 1;
19          END;
20          PrevDate := ItemLedgerEntry."Posting Date";
21          TotalQty := TotalQty + ItemLedgerEntry.Quantity;
22        UNTIL (ItemLedgerEntry.NEXT = 0) OR (count >= 4);
23        IF PrevDate <> 0D THEN BEGIN
24          OutputData(1, Item."No.",Item.Description,PrevDate,-TotalQty);
25          count := count +1;
26        END;
27    UNTIL (Item.NEXT = 0) OR (count >= 4);
```

Figure 1

The above example calculates information about Item Movement. Because we are interested in knowing the total quantity sold per date per item. It may not be obvious at first what fields are selected, which tables are involved, etc. If you look more closely you will note:

- Line 11 describes the link criteria of the two tables ('Item' and 'Item Ledger Entry'): for an item, gather all the existing Item Ledger Entries, matching on item number.
- Line 10 specifies the ordering of the result set: order first by item number, then by posting date. This ordering is important as it allows us to compare the Posting Date on the current record with the one from the previous record in order to detect when this value changes (see Line 15).
- Line 12 applies a filter for only records of type sales.
- Line 21 Sums the quantities on all lines for a particular date and item.
- Line 22 and 27 ensures that only the first five entries are recorded.
- Lines 14 and 22 put in the necessary looping for summing.
- Lines 7 and 27 put in the necessary looping for each item.

Now, let us compare the above implementation with an implementation utilizing the new query object. I have created a simple query to retrieve the same desired data. As you can see in the following illustration, the desired tables and subset of fields are apparent from a quick glance. The summation per date per item is also clear by inspecting the 'Group By' column.

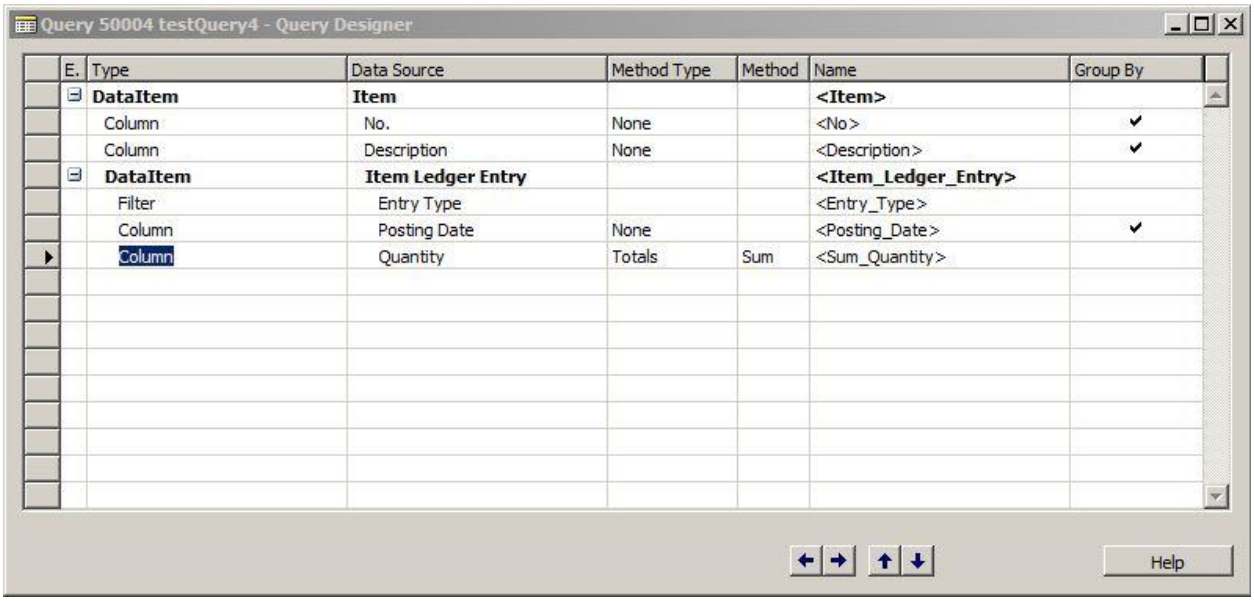

Figure 2

Now, since we still want to use this calculation from code, let us look at the revised code snippet.

```
1   //ItemMovements is a variable of Type = Query and SubType= testQuery4 (above)
2   ItemMovements.TOPNUMBEROFROWS := 5;
```


© 2012 Microsoft Corporation. All rights reserved.

```
3       ItemMovements.SETRANGE(Entry_Type,ItemLedgerEntry."Entry Type"::Sale);
4       ItemMovements.OPEN;
5       WHILE ItemMovements.READ DO
6        OutputData(2, ItemMovements.ItemNo, ItemMovements.Description,
7              ItemMovements.PostingDate,ItemMovements.Sum_Quantity);
8
```

Figure 3

The code in Figure 3 is much shorter, much easier to read, and much easier to explain than the code in Figure 1:

- Line 2 specifies the desire for only five items
- Line 3 applies the necessary filters.
- Lines 4 through 8 execute the query and loop through the result set.

While I did state that functionally you could accomplish the same goals using Records or Queries, there are a few cases where the result sets may differ slightly. Query result sets are not guaranteed to be dynamic, meaning that the result sets may (or may not) contain the changes made to the database records over the course of the current transaction. In other words, if you modify a record, call it A, and you happen to read parts of record A as a row in your query result set later on, you may see the changes or you may not. Utilizing the Record API, you would always see the changes because the Record API guarantees dynamic results. Therefore, the values of the rows retrieved from the Query and Record APIs may differ slightly, depending on the changes made during the current transaction.

Besides simplifying code that is written, queries can be used in C/AL as datapumps when there is a need to read/display/extract joined, filtered, or aggregated data in a ***performant*** manner. Not only can they be applied to read-only scenarios, but you can also extract a set of records that need to be modified. Note that the actual modification still needs to be done via the record API. Depending on the 'ReadState' property (described immediately below) of the Query, you may need to explicitly read the row again utilizing the record API (consider using the 'ReadExclusive' state so that you do not require an additional read).

ReadState

The ReadState property of the Query object briefly mentioned above determines the level of concurrency which will be allowed for the SQL Server SELECT statement that is generated by running the Query object. For example, this property can be used to control whether or not the Query is allowed to ignore any SQL Server locks as it reads through records in the database. The following ReadState Query property values are available:

ReadUncommitted

Reads all data in the database, regardless of whether the entry has been committed, i.e. regardless of whether the data has been saved to the database upon completion of the transaction. This means that the query can read data from table rows that have been modified by another running transaction (i.e. user), but have not yet been committed.

This mode is sometimes referred to as a "dirty read". For those familiar with SQL, this translates to reading the data with the 'Read Uncommitted' isolation level.

No additional locks are placed on the data as it is read and any existing locks (from other transactions) are ignored.

Setting the ReadState property to **ReadUncommitted** can improve performance compared to **ReadShared** and **ReadExclusive**. However, the query could read data that may be subsequently rolled back or changed by another transaction while the query is running. Therefore, ReadUncommitted would not be a good choice if the Query is being used to read records which are then going to be updated using the Record API because of the risk that the data could change after being read by the Query and before being re-read by the Record API.

ReadShared

The ReadShared option will ensure that only committed data is read and selected for the query result set.

SQL Server "Share Locks" are placed on all data as it is read which prevents it from being modified or deleted by other transactions. The locks are held until the current transaction is committed. However, it is possible for other transactions to read data which has a Share Lock placed on it as long as the other transaction does not require an Exclusive Lock, e.g. other transactions which require Share Locks or transactions which ignore all locks (i.e. dirty reads).

If you are familiar with SQL, this mode translates to reading the data with the 'Repeatable Read' isolation level.

ReadExclusive

© 2012 Microsoft Corporation. All rights reserved.

The ReadExclusive option will ensure that only committed data is read and selected for the query result set.

SQL Server "Update Locks" are placed on all data as it is read which prevents it from being modified or deleted by other transactions. While existing data is locked in this way there is still the possibility that other users could insert new rows within the same key range which is being read by the Query. Also, it may be possible for other users to read the locked data depending on their transaction types.

For readers familiar with SQL Server, this mode translates to reading the data with the UpdLocks table hint.

The ReadState property will apply to the SELECT statement executed by the Query regardless of the current transaction type as set by a CURRENTTRANSACTIONTYPE Function call in the C/AL code. This is because Queries are not affected by the CURRENTTRANSACTIONTYPE function call which only applies to data being accessed with the Record API.

Each Query will use the specified ReadState regardless of other Queries that have already been executed. This means that you can attempt to read uncommitted data and committed data from the same tables in the same transaction. However, the strictest lock placed on a row will remain until the transaction is committed. Therefore, if a ReadExclusive Query runs in the same transaction as a ReadUncommitted Query and both access the same set of records then effectively both Queries will be treated as ReadExclusive.

Reports

Moving on to the Query object's sibling, the Report object, Microsoft Dynamics NAV Reports consist of two pieces:

- The expression of what data to gather (the Report Designer)
- The actual visual representation of this gathered data (report layout)

Since queries do not have a visual piece, this comparison will be solely made from the perspective of the backend engine that gathers data. Unlike the query, the backend engine is not completely aligned with SQL Server to retrieve the data for reports, due to the flexibility of functionality. Through triggers on each of the selected DataItems, the execution of data retrieval can be manipulated dynamically at runtime for the reporting stack. With the addition of such code, it is not possible to turn the modeled report object into a single SQL select statement, similar to the query object. All of the linking, totaling, grouping or filtering must be performed on the NST as opposed to the SQL server. With this implementation of the stack, potentially more data is retrieved and transferred from SQL server to NST, before eventually being condensed with filtering and totaling, as opposed to, condensing during retrieval and only transferring what is desired. Since all the calculations must be done on the NST on potentially much larger datasets, the query object outperforms its predecessor in the structured cases that it can encompass. This is another clear example of the tradeoff between flexibility in functionality and optimizing for performance.

Though it is not possible to directly bind a report to a query in Microsoft Dynamics NAV 2013 (allowing the report to be the visual representation and the underlying query to be the backend data retrieval engine), there is a simple workaround, which will provide most of the desired functionality. For some reports, the following approach can be used:

1. Insert a DataItem bound to an Integer data source.
2. Inside the OnPreDataItem trigger, run a Query which has been designed for the purpose of retrieving the required data (<queryVar>.OPEN)
3. Then, inside the OnAfterGetRecord trigger loop through the resultset (using <queryVar>.READ while making sure that it returns 'TRUE' to avoid an endless loop)
4. Finally, in the OnPostDataItem trigger, call <queryVar>.CLOSE to ensure disposal of the enumerator

Through this process, it will be possible to speed up retrieval of some nodes in the report.

XMLports

There are several similarities and differences between queries and XMLports. Both queries and XMLports allow you to export data from the database. While XMLports allow you to also import data, we will not discuss this functionality, since queries do not have a counterpart.

© 2012 Microsoft Corporation. All rights reserved.

XMLport object is the static formatter of data, which allows similar runtime manipulation of the data collected as the Reporting stack. The ability to aggregate, order, or group without having to write C/AL code is limited. Doing such processing in code in XMLports is slow for the same reasons as it is for reports and potentially could be done in SQL. For scenarios that can be modeled as a Query, the entire desired dataset can be retrieved from the database with an underlying single SQL select statement and exported using the C/AL SaveAsXML or SaveAsCSV methods.

Queries differ the most from XMLports in scenarios where dynamic processing is needed in manipulating the resulting dataset, such as user-specified filters. For XMLports, the processing must be done within the object in C/AL code and is applied to the data on the NST; while for queries, it can be applied on top of the modeled object and to the underlying select statement executed on the SQL Server. The ability to apply filtering outside the Query object in this way encourages reusability. Also, the underlying stack is optimized to handle such dynamic processing efficiently.

We hope that Microsoft Dynamics NAV partners will use queries in key application scenarios that they implement in Microsoft Dynamics NAV 2013, which require large amounts of data retrieval and where performance is critical. We have taken advantage of this object in key places within the Microsoft Dynamics NAV standard application, such as the Trailing Sales Order Chart on the Sales Order Processor Role Center, the Lot Numbers By Bin Fact Box on Warehouse Picks, and Report 19: VAT- VIES Declaration Tax Auth to name a few examples. As Microsoft Dynamics NAV partners use this new object in their code, we also plan to continue to use it within the standard application code in future releases.

Lesson 2: Linking Subtleties

In Lesson 1, we introduced the new Microsoft Dynamics NAV Query Object into the Microsoft Dynamics NAV Development Environment, comparing it to a few of its sibling Objects and describing when you should take advantage of its functionality. Now that you feel comfortable about using Queries, let us dive deeper into some of the subtleties of designing more complicated queries.

One of the first things that you will try out after the equivalent "Hello World" query of retrieving a few fields from a single table is to retrieve data from multiple tables. Working in the Query Designer, you start by adding a few data sources (different tables) and selecting a subset of fields, similar to Figure 4 below.

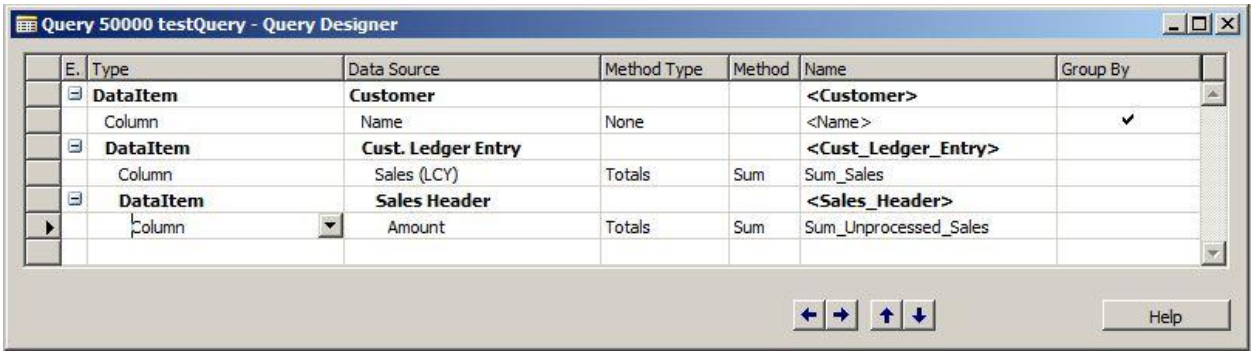

Figure 4:

If you try to compile the above Query without modifying any properties, you would get an error messaging stating that you need to specify an expression for the DataItemLink property. Ideally, you should also consider the DataItemLinkType property at this time, although a default value is provided.

We provide five different types of linking, aligned with the Join types in SQL Server. If you are familiar with SQL, the equivalent SQL Join types would be:

- Inner Join
- Left Outer Join
- Right Outer Join
- Full Outer Join
- Cross Join

If you are not familiar with SQL join types, we have surfaced two of the more common types:

- **Exclude Row if No Match.** If a row in the first table has no matching row in the linked table based on the linking criteria then neither row is returned in the result set. This is the equivalent to a SQL Inner Join

© 2012 Microsoft Corporation. All rights reserved.

- **Use Default Values if No Match.** If a row in the first table (i.e. the "left" table) has no matching row in the linked table (i.e. the "right" table) then the row from the first table will be returned along with default values (e.g. zero for integers, empty string for text fields etc.) for all columns associated with the linked table. This is the equivalent to a SQL Left Outer Join

References and further reading material:

- *Types of SQL Joins*
- 'Understanding Data Item Links' article in the Microsoft Dynamics NAV Developer and IT Pro Help.

Let us look further into the query shown in Figure 4. It looks like it is ***trying*** to retrieve the sum of processed sales (Sum_Sales column) and the sum of unprocessed sales (Sum_Unprocessed_Sales column) per customer. Note the word "trying" in the previous sentence, this is used because the designed query does not retrieve what you may expect. For this example, assume that the DataItemLinkType of both DataItems is 'Use Default Values if No Match' (equivalent to a SQL Left Outer Join) and that the DataItemLinks have been set up properly on the Customer Number.

At first glance, you may expect the output of this query to be a result set containing each sum per customer. By choosing the 'Use Default Values if No Match' option, we assume that all customers are returned regardless of the amount of their processed and unprocessed sales. However, the output is not as expected, since the underlying query implementation is aligned with SQL, which has a very specific (and sometimes counter intuitive) way of joining tables. To understand what result set is returned by the query, we need to dig a bit deeper into the SQL.

For this example, let us assume that we have the following tables.

Customer Table

Customer No.	Customer Name
10000	The Cannon Group PLC
20000	Selangorian Ltd.

Cust. Ledger Entry Table

Entry No.	Customer No.	Sales (LCY)
1	10000	1500
2	10000	4500

Entry No.	Customer No.	Sales (LCY)
3	20000	3500
4	20000	5500

Sales Header Table

Entry No.	Customer No.	Sales (LCY)
1	10000	1000
2	10000	4000
3	10000	6000
4	20000	3000
5	20000	2000
6	20000	5000

If you disregard optimizations and think conceptually, SQL joins the Customer table to the Cust. Ledger Entry table by matching on the Customer No. fields. So, in our example, The Cannon Group PLC (Customer No. 10000) has two records in the Cust. Ledger Entry table (Entry No. 1 and 2) with Sales (LCY) of 1500 and 4500. Selangorian Ltd. also has two entries (3 and 4) with Sales of 3500 and 5500. This can also be seen in the first two columns of the diagram shown in Figure 5 . Note, we have excluded the other columns to make the diagram easier to read, however the field selection operation will happen at a later point in time on SQL Server.

Our Query also joins the Customer table to a second linked table (i.e. the Sales Header table) by matching on the Customer No. fields. So, in our example, The Cannon Group PLC has 3 entries (1, 2, and 3), with Sales of 1000, 4000, and 6,000. Therefore, **for each row** returned from the first linked table (i.e. the rows with Sales of 1500 and 4500) there are now **three rows** returned from the second linked table. Selangorian Ltd. also has three entries (4, 5, and 6) in the second linked table, with sales of 3000, 2000, and 5000. For each row in the first linked table (i.e. the rows with Sales values of 3500 and 5500), we will have three rows returned from the second linked table. Therefore, a total of 12 rows will be produced in the intermediate result set for this query (before aggregation is applied) showing the Sales values in the third column in Figure 5:

© 2012 Microsoft Corporation. All rights reserved.

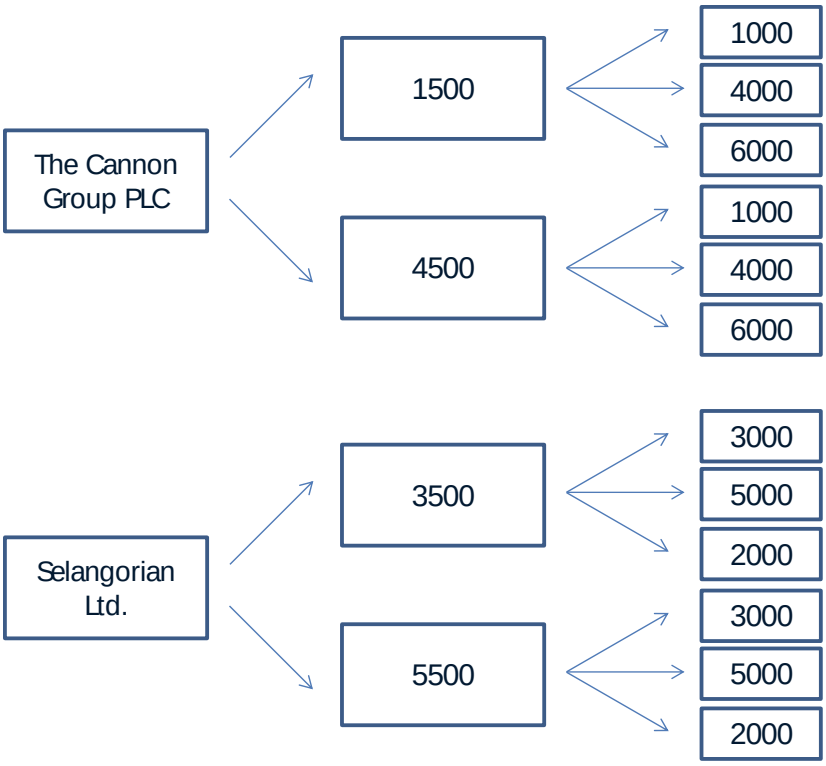

Figure 5

Therefore, the intermediate result set generated for our Query object (before aggregation is applied) would be as follows:

Customer Name	Sales (LCY) from Cust. Ledger Entry	Sales (LCY) from Sales Header
The Cannon Group PLC	1500	1000
The Cannon Group PLC	1500	4000
The Cannon Group PLC	1500	6000
The Cannon Group PLC	4500	1000
The Cannon Group PLC	4500	4000
The Cannon Group PLC	4500	6000
Selangorian Ltd.	3500	3000
Selangorian Ltd.	3500	5000
Selangorian Ltd.	3500	2000
Selangorian Ltd.	5500	3000

Customer Name	Sales (LCY) from Cust. Ledger Entry	Sales (LCY) from Sales Header
Selangorian Ltd.	5500	5000
Selangorian Ltd.	5500	2000

Now, if we calculate the Sum on 'Sale (LCY)' from the Cust. Ledger Entry table per customer, we get:

- The Cannon Group PLC : 1500*3 + 4500*3 = 18,000
- Selangorian Ltd: 3500*3 + 5500*3 = 27,000

Similarly, if we calculate the Sum on 'Sales (LCY)' from the Sales Header table per customer, we get:

- The Cannon Group PLC : 1000*2 + 4000*2 + 6000*2 = 22,000
- Selangorian Ltd: 3000*2 + 5000*2 + 2000*2 = 20,000

Note, the 'Sale (LCY)' from the Cust. Ledger Entry table numbers have been repeated three times, since there were three matching entries in the Sales Header table and the 'Sales (LCY)' from the Sales Header table numbers have been repeated twice, since there were two matching in entries in Cust. Ledger Entry table.

This occurs because of the order in which SQL executes different clauses of the select statement (remember, the statement that retrieves the desired data). As we discussed in the above example, the SQL Server query engine first retrieves the data from each table, linking it to the other tables. From this intermediate result set, it then eventually aggregates the values grouped by the desired fields and finally selects the requested fields.

The following shows the order of execution for each clause of the relevant SQL Server query:

1. FROM clause is executed first (DataItems selected)
2. ON (DataItemLink)
3. JOIN (Indentation)
4. WHERE (Filtering options)
5. GROUP BY
6. HAVING (Filters on Totaling Columns where applicable)
7. SELECT (Columns selected)
8. ORDER BY

© 2012 Microsoft Corporation. All rights reserved.

9. TOP (TOPNUMBEROFROWS)

References and further reading material:

- *SQL Server SELECT statement*

So, now it seems that we are stuck, but let us not give up so soon. Let us revisit the same example, but try to take advantage of another Microsoft Dynamics NAV concept, FlowFields. As you may recall, on the Customer table, there is actually a FlowField defined that sums the Cust. Ledger Entries for each customer. This FlowField is called 'Sales(LCY)' and encapsulates the SUM of the SALES(LCY) portion of the query.

Figure 6 shows what the resulting modeled Query looks like if we redesign it to utilize the 'Sales (LCY)' FlowField.

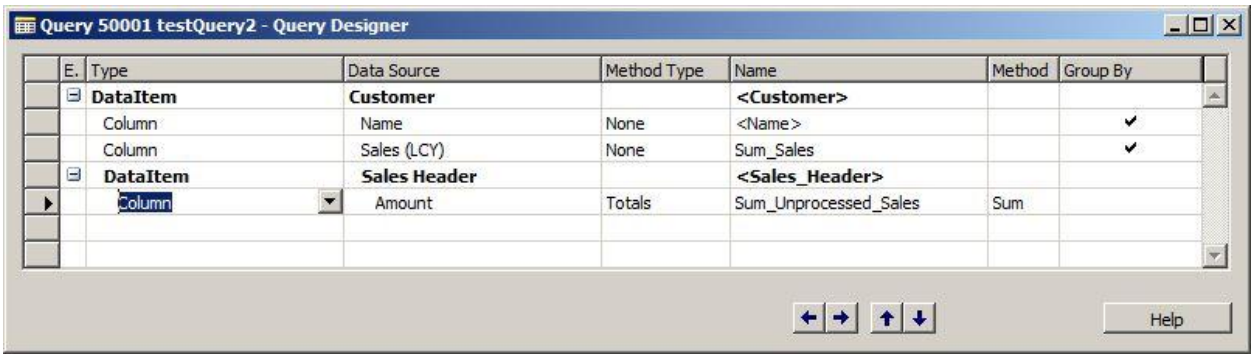

Figure 6

This query now produces what was originally expected by the first query. The reason this new Query object works as expected is that one of the sums is produced via a SQL sub-query, namely the one for the FlowField. In short, without going into the details of the concept of sub-queries, you can think of the FlowField as an entirely separate query that produces a result, which is treated like a normal column value by the SELECT statement in the main query.

Currently in Microsoft Dynamics NAV 2013, the query object does not support SQL Server sub-queries, except through FlowFields as described above.

Lesson 3: Filtering Subtleties

There are different types of filtering for different purposes. Before discussing any subtleties, let us review the different types and their usage.

On each DataItem, there is a property called 'DataItemTableFilter'. In this property, you can select any field from the underlying source table even if it is not one of columns defined in the SELECT statement, i.e. even if the field is not displayed as part of the Query result set. This property's value is part of the internal details of the query and is not exposed externally through the Query C/AL API. As such, it is an immutable filter, meaning it cannot be changed dynamically at runtime or overwritten by an end user.

On each column, there is a property called 'ColumnFilter'. In this property, you can specify any filter expression on the column. This filter setting is mutable which means it can be changed dynamically, e.g. in C/AL code. Since this a filter on the column and not on the underlying source table, the filter applies to the resulting value. Therefore, if the "Reverse Sign" property is selected for the same column or if a "Date" or "Totals" method is applied, the ColumnFilter property will apply after this additional computation. For example, imagine you have bound query column "A" to an integer source field and you have also enabled the ReverseSign property for this column. You then apply a ColumnFilter on column "A" so that it will only retrieve positive values. In this scenario, the only values which would be returned would be where the original source value was a negative number. This is because only numbers which are negative in the source table would be positive after the "ReverseSign" operation has been applied and therefore only these would qualify according to the filter expression.

Besides adding rows of type column to a query, you can add rows of type 'Filter'. This construct is used to surface fields that need to be filtered on dynamically (either through the C/AL API or by the end user) without being selected as part of the result set. Thus, filters applied in the 'ColumnFilter' property on 'Filter' columns are mutable, meaning they can be overwritten dynamically at runtime. As already mentioned, they are not included in the result set. For SQL experts, this means they are not included in the "SELECT" and "GROUP BY" clauses (when applicable).

All three types of filters are done after the linking. The order of execution is critical, as you will see in the following example. If you are familiar with SQL, these filters are placed in the "WHERE" or "HAVING" clause of the SQL SELECT statement and NOT as part of the join condition (in the "ON" clause). The placement of the filter, in the SQL "WHERE" or "HAVING" clause, depends on whether there is a totaling method applied to the query.

Let us now look at an example that will make the subtleties of filtering more apparent. The following screenshot shows a simple query to find the sum of unprocessed sales per salesperson, similar to the previous example we were using in the above "Linking Subtleties" Lesson. Assume the DataItemLinkType property is set to 'Use Default Values if No Match' (i.e. a SQL Left Outer Join) and that the link between the two tables is set up

© 2012 Microsoft Corporation. All rights reserved.

properly. By choosing 'Use Default Values if No Match', you want to get a list of all salespeople regardless of the amount of their sales.

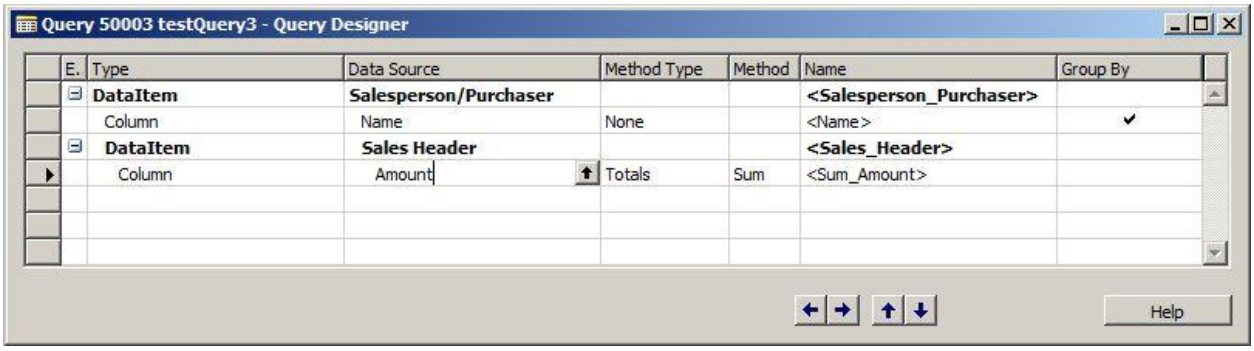

Figure 7

Let us say that you decide to place a filter on the 'Order Date' since you are only interested in sales in 2012. Note, instead of placing the filter in DataItemTableFilter, you could add a row of type 'Filter' with the data source bound to the 'Order Date' field. This example will be identical, regardless of the location in which the filter is placed.

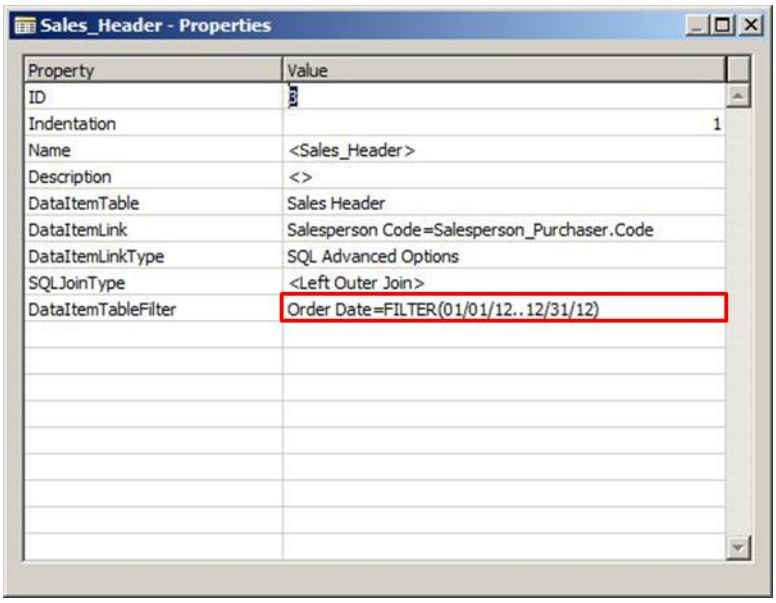

Figure 8

One may think that the result set of this query contains the sum of unprocessed sales per salesperson for 2012 for all sales people. As you may have guessed, this is not the case, since why then would I be describing subtleties of filtering with this example.

For this example, let us assume that we have the following tables.

Salesperson/Purchaser Table

Code	Name
JR	John Roberts
PS	Peter Saddow

Sales Header Table

No.	Salesperson Code	Amount	Order Date
1	JR	1000	2/12/2012
2	JR	4000	4/16/2012
3	JR	6000	8/13/2011
4	PS	3000	6/15/2011
5	PS	9000	9/22/2011

If you disregard optimizations and think conceptually, SQL first joins the Salesperson/Purchaser table to the Sales Header table by matching salesperson codes. Therefore, in our example, John Roberts has three entries (1, 2, and 3) with Sales of 1000, 4000, and 6000. Peter Saddow has two entries (4 and 5) with Sales of 3000 and 9000. This can also be seen in the first two columns of Figure 12 below. Note, we have included the 'Order Date' as a third column in dashed boxes, simply to easily see the values for this field for each row.

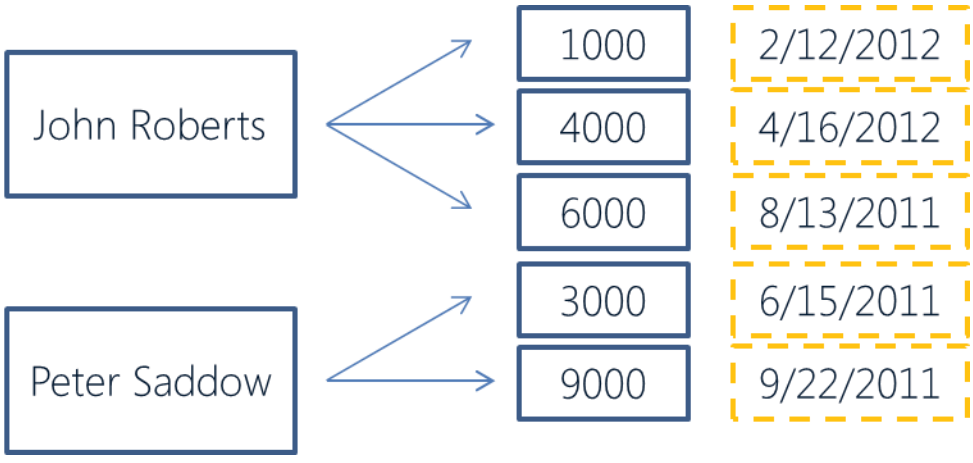

Figure 9

© 2012 Microsoft Corporation. All rights reserved.

If we were to translate the diagram into a flat intermediate result set, we would get the following table:

Salesperson Name	Total Sales	Order Date
John Roberts	1000	2/12/2012
John Roberts	4000	4/16/2012
John Roberts	6000	8/13/2011
Peter Saddow	3000	6/15/2011
Peter Saddow	9000	9/22/2011

Now, because of the DataItemTableFilter property (see Figure 8), SQL filters out all dates not in 2012, thereby losing the last three rows in the intermediate result set above. Thus, we are left with two rows, both for John Roberts. These two rows then get aggregated (summed) into a single row because we specified aggregating per salesperson in Figure 7. The final result set can be seen below and does not contain any entry for Peter Saddow despite the fact that we set DataItemLinkType property to 'Use Default Values if No Match'.

Salesperson Name	Total Sales
John Roberts	5000

The behavior described above occurs because the order in which SQL executes different clauses of the Select statement (remember, the statement that retrieves the desired data). As we went through in the example, SQL Server query engine first retrieves the data from each table, linking it to the other tables. From this intermediate result set, it then filters out unwanted entries and then finally aggregates the values grouped by the desired fields to select the requested fields.

References and further reading material:

- *SQL Server SELECT Statement*

Using SQL Server directly we could avoid this problem by adding the filtering criteria specified in Figure 10 above into the Join criteria for the SELECT statement. The equivalent in a Microsoft Dynamics NAV Query object would be to integrate the filter criteria into the DataItemLink property. This functionality is not supported in Microsoft Dynamics NAV 2013, but may be included in a future release.

Lesson 4: Implementing Queries for Adhoc Reporting and BI

In this lesson, we explore the considerations that you need to keep in mind when you create Queries that will be exposed through OData for ad hoc reporting solutions and those that will be used for charting with the Chart Configuration Page.

OData Enabled Queries

A full description of the OData protocol is beyond the scope of this Readiness training. However, this section will describe how a Query object can be used as a data feed by publishing it as a web service through OData. One useful scenario where queries as data feeds can be utilized is through PowerPivot, to allow customers to slice and dice their data to analyze trends and help make business decisions. This lesson is not focused on all the scenarios where OData enabled queries can be used, but on what you as a developer should consider when creating queries for these scenarios.

We will now walk through the restrictions that have been placed on queries that can be exposed as web services through OData. This information will also provide some insight into the motivation for these restrictions.

TopNumberOfRows and $top

By this point, you are hopefully aware of the TopNumberOfRows property in the main properties of a Query object as seen in Figure 15 below. This property specifies what number of rows to return from the resulting dataset. The default value for this property is that all rows in the resulting set are returned. If a value is specified, say X, then the first X rows are returned. If the number of rows in the entire result set is smaller than the value set for this property then the entire result set is returned. This property can also be changed dynamically through C/AL.

© 2012 Microsoft Corporation. All rights reserved.

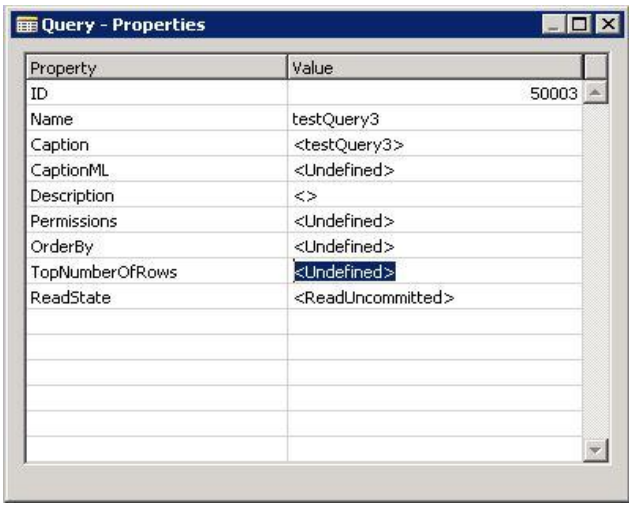

Figure 10

If you would like to expose a query as a web service, the value specified in the TopNumberOfRows property in the designer or through C/AL, must be less than or equal to the OData paging size (or full set, no TOP value) specified in the server configuration. Server side paging has been implemented for this protocol to ensure that queries producing large result sets can still be handled. A result set with more rows than the Odata paging size requires multiple calls by the framework to retrieve the complete set. This is generally possible since OData returns a token of where to continue retrieving rows. However, there is a key difference with result sets producing more rows than the paging size when there is a TOP clause. Somehow the number of rows which still needed to be fetched (Value specified in TOP- Odata Page Size * Number of Calls Made up to this Point) would need to be kept. For example, after the first page of data, we would need (Value specified in TOP - OData Paging Size) more rows. Since OData is a stateless protocol, there is no way to keep track of the number of rows to still retrieve.

A default value has been configured for the OData paging size in Microsoft Dynamics NAV 2013. One workaround is to increase this value for a customer if you need to implement a query that requires the top number of rows to be larger than the OData paging size. We state this with caution, since a paging size has been specified for a reason, i.e. to protect against large result sets that consume all the memory on the server.

Now, let us look at $top which is the OData counterpart of Microsoft Dynamics NAV TopNumberOfRows Query property. If a value is specified in the OData URL, it will work for the end user, but may not be what is expected. The value for $top is applied post processing, i.e. after the execution of the query. Therefore, the number of rows in the result set seen at the consumer end will be the lesser of $top and TopNumberOfRows. In other words, the value in $top cannot have the effect of increasing the value specified in TopNumberOfRows property of the Query object.

For a standard installation, the URL described above would look like:

*http://localhost:7048/dynamicsnav70/OData/**myExposedQueryName**?$top=**SomeValue***

OrderBy and $orderby

In Microsoft Dynamics NAV, specifying a unique ordering for the result set of a query is not required. However, OData requires that the result set be uniquely ordered to guarantee proper paging of the retrieved data. A unique ordering is not enforced, but will be created automatically by the query runtime if one is not provided.

While specifying a value for the OrderBy property on a query works as expected when the query is exposed as a web service, there are considerations when an additional $orderby is specified by the end user in the OData URL. If a value for $orderby is specified, it will work only if the resulting dataset of the query is less than or equal to the OData paging size that is specified in the server configuration.

For a standard installation, the URL described above would look like:

*http://localhost:7048/dynamicsnav70/OData/**myExposedQueryName**?$orderby=**SomeColumnName***

The ordering specified through $orderby is applied as a post processing, so the result is re-ordered after the query's result set has been returned. If the result set from the query is larger than the OData paging size, the ordering specified by the user would be per page and not across the entire result set. To avoid confusing end users, in Microsoft Dynamics NAV 2013 we have restricted this functionality to work only if the Query result set size is within the OData page size.

The workaround may be to increase the paging size for a customer if you need to implement a query that requires the end user to order (post-processing) on a result set larger than the OData paging size. The same warning that was mentioned above applies, i.e. the default paging size was specified to protect against large result sets that consume all the memory on the server.

The preferred workaround if the ordering of the query is not sufficient and the result set is potentially large is to simply order the result set after the end user has retrieved all the data. For example, a customer may reorder the data in Microsoft Excel after he/she has imported the entire result set.

$filter

Unlike for charts and in C/AL, if a value for $filter is specified by an end user in the OData URL, it will not overwrite ColumnFilters defined in the query. This filter is applied post

© 2012 Microsoft Corporation. All rights reserved.

processing like the other operators described above, so logically the additional filters are AND'ed together.

For example, suppose you have created a Query object with a column with an underlying source field called City. On this column, you have populated the ColumnFilter property to filter to Chicago (City = Chicago). The following screenshot shows an example query.

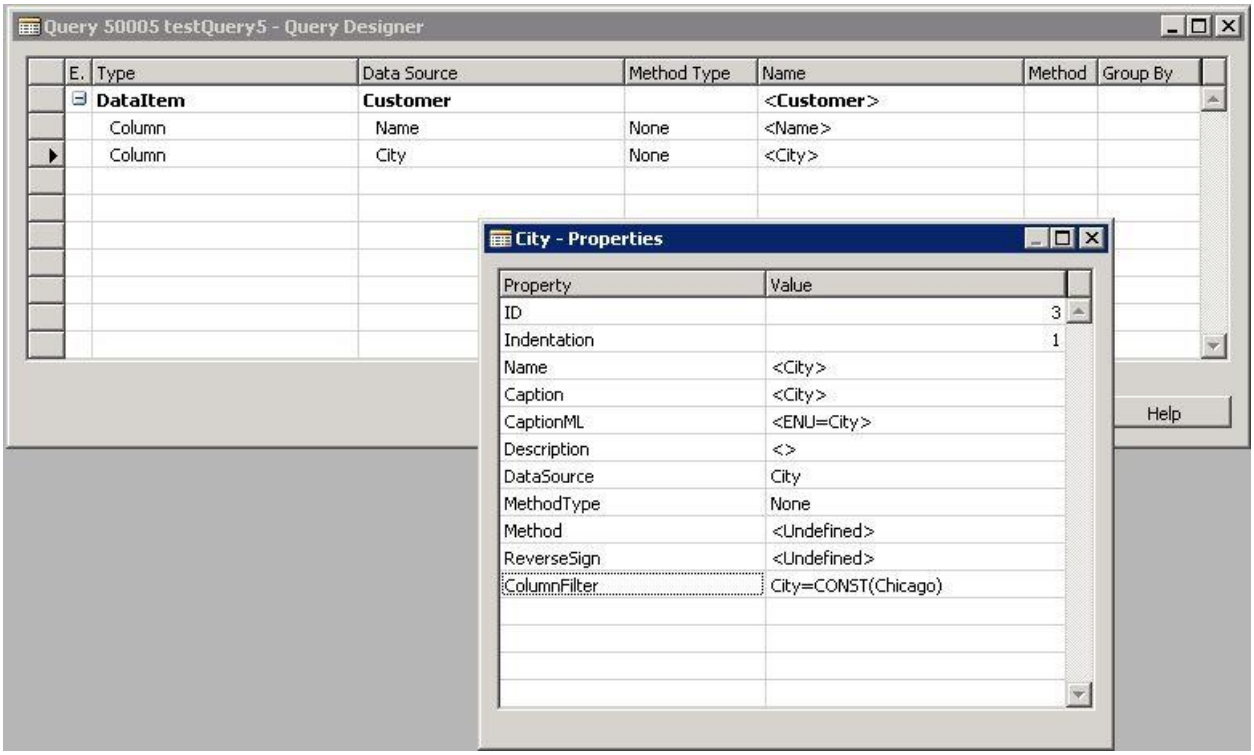

Figure 11

Then the end user adds a filter City = Miami specified in the OData URL. For a standard installation, the URL described above would look like:

*http://localhost:7048/dynamicsnav70/OData/**myExposedQueryName**?$filter=**Miami***

The result set will always be empty for this query exposed through OData, since no records in the Customer table have a city location equal to both Chicago and Miami. However, with Microsoft Dynamics NAV Charts, the filters would be overwritten if placed in the ColumnFilter property and the customers located in Miami will display. More information on Charting will be covered later in this lesson.

Queries for Charting

With Microsoft Dynamics NAV 2013, we have improved the charting story significantly to allow end users to create their own charts. There is no more need to write XML by hand to

create charts bound to a single table. We have introduced the Chart Configuration Page that allows end users to create Charts bound to tables and queries. Figure 12 shows this new Page:

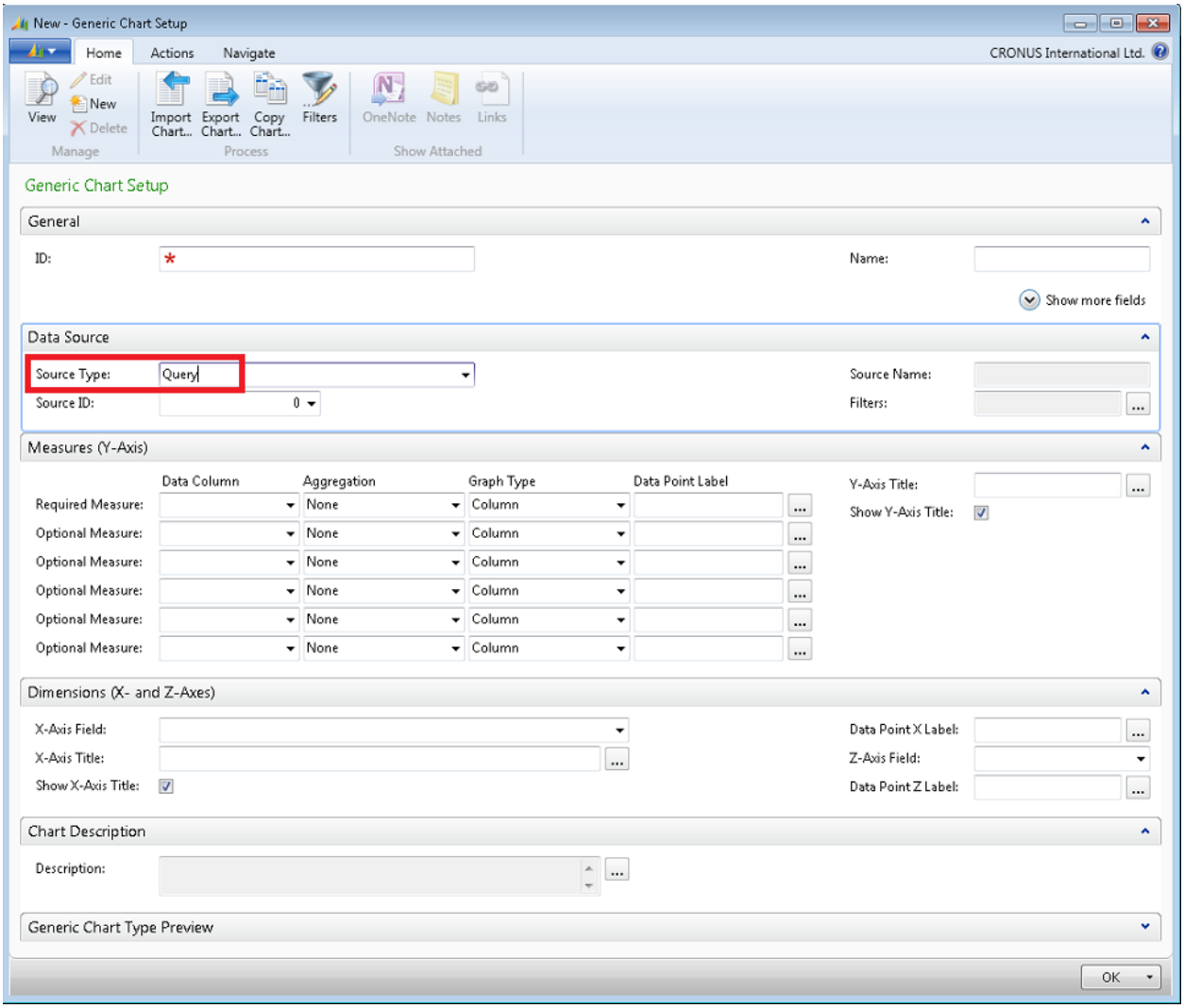

Figure 12

More information about the Chart Configuration Page will be provided in the Microsoft Dynamics NAV 2013 product documentation. In this lesson we want to describe tips and tricks that you should consider when you create a Query object that is used is to power a Chart.

First and foremost, it is important that your customers, the end users, are aware of what the provided Queries object can actually display. Out of the box, all Queries are included in the look up of the Source ID in the Chart Configuration Page. However, if you end up creating a significant number of Queries, it may be difficult for an end user to decide which Queries are intended for Charting and which are not. The Chart Configuration Page is an ordinary application page so one could modify it to filter out Queries that should not be a source for a

© 2012 Microsoft Corporation. All rights reserved.

chart. It is important that customers are provided with a list of Queries they can use and a description of the "view" it provides. If they are not informed, they may end up creating charts that show something other than what they expected.

Through the Chart Configuration Page, end users can choose a subset of the columns selected in a Microsoft Dynamics NAV query, change the totaling method on any Column whose underlying data source has type Decimal, and add additional filtering on Columns.

Let us revisit a query that we discussed in Lesson 1:

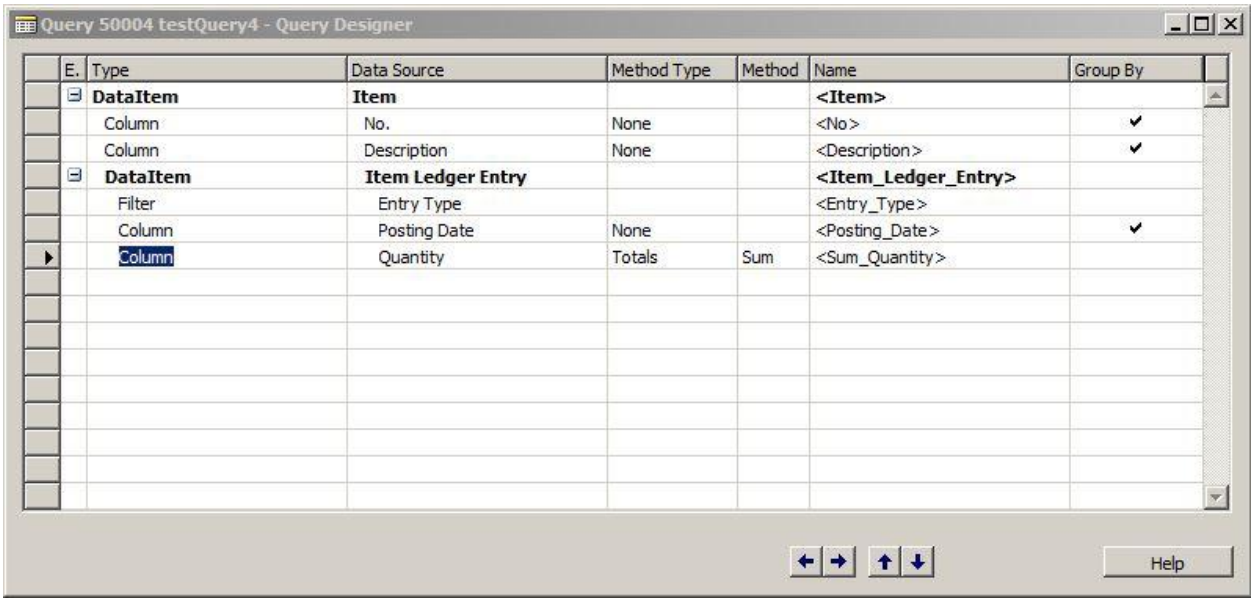

Figure 13

Suppose an end user decides to create a Chart bound to this Query object, selecting the column 'No' as the dimension (Y-axis) and the column 'Sum_Quantity' as the measure (X-axis). A question can arise about how the 'Posting Date' field should be treated. The user expects a sum of quantity for each item number. However, as the query is defined, the user would get the sum of quantity per posting date for each item number. Instead of simply removing the posting date post-processing, which is clearly not what the end user expects, the charting runtime turns the 'Posting Date' column into a row of type 'Filter'. Thus, the courser aggregation is accomplished without losing the potentially referenced 'Column'. Any filters applied to this column are still applied. Any code accessing this column in C/AL code to set a filter does not fail either. If this column had been part of the OrderBy property, it would be simply removed. For charts, the order of data may not even be as relevant.

Suppose now the end user decides to change the totaling methods for a column. In our example, suppose he/she wants to make the sum an average. Captions such as 'SumOfQuantity' for a column that is a SUM on the 'Quantity' field may no longer make sense. While end users may choose to add a custom caption in the Chart Configuration Page, it may still be useful to use generic names such as 'AggregatedQuantity' to avoid confusion when creating queries meant specifically for charting.

Finally, a word of caution about specifying a filter using the ColumnFilter property when designing a Query object. An important point about this property is that it is mutable, i.e. it can be changed/overwritten by the end-user at runtime. Therefore, if a ColumnFilter property is saved as part of the Query object at design time and subsequently, at runtime, the end user provides a different value for the ColumnFilter then the value provided at runtime will overwrite the original value specified with the Object definition. Therefore, Query designers should avoid specifying a ColumnFilter property where there is a risk that overwriting this value can cause problems.

Of course, filters applied in the DataItem property can never be overwritten as they are immutable. Therefore, any relevant user-defined filters applied at run-time will be appended to the Query's DataItem property filters, (i.e. a logical AND is used to append the filters in the Query DataItem property and the user defined filters specified at runtime).

Note that User defined filtering specified at runtime does NOT overwrite the filters set on that column using SetFilter/SetRange in triggers. On the contrary, SetFilter/SetRange statements are applied in triggers which are executed after the end user has applied any filters. Therefore the SetFilter and SetRange filters will overwrite the end-user filters, i.e. they will not be appended to the end-user filters.

© 2012 Microsoft Corporation. All rights reserved.

Lesson 5: Query Objects and Performance

This lesson describes how to design queries and table keys in the most efficient way.

FlowFields in Queries

Query objects manage FlowFields by including a subquery in the SQL Select statement to retrieve the appropriate FlowField value thus enabling the Query to retrieve all of the data in a single request.

Covering Indexes

When you use a query to select a subset of fields in a table, you should considering taking advantage of the covering index strategy. A *covering index* is an index that contains all output fields required by the operation performed on that index. A covering index data access strategy can vastly improve performance because the database engine can retrieve the required data from the index directly without needing to do any I/O against the underlying table. A covering index data access strategy can be used when the following conditions are true for the relevant DataItem:

- All columns in the relevant DataItem must be part of a single Microsoft Dynamics NAV key.
- All columns used in the relevant DataItem's table filters are also part of the same Microsoft Dynamics NAV key.
- All columns used to link the relevant DataItem to other DataItems in the same Query Object must also be part of the same Microsoft Dynamics NAV key.

The SQL Server optimizer will automatically choose a covering index strategy whenever possible.

Covering SIFT Indexes

SIFT indexes can also be used to retrieve data for a query that contains totals. SIFT totals are maintained after each insert, modify, or delete call, and so some or all of the commonly used totals are pre-calculated. A covering SIFT index can be used when the following conditions are true:

- The query contains at least one aggregated column with Method Type set to Totals and with Method set to Sum, Count, or Average.

- If a DataItem contains an aggregated column, then all columns under that DataItem must be aggregated columns, must use the Sum, Count, or Average method, and must be part of a SumIndexField defined on a single Microsoft Dynamics NAV key.
- In a Query in which you have aggregations on some (but not all) DataItems, then for the DataItems without aggregations, the relevant columns from the Query must be part of a SumIndexField.
- All non-aggregated columns under the DataItem which has the aggregation values are part of the key fields defined for the relevant SIFT index.
- All columns that are used in the relevant DataItem table filters are part of the same Microsoft Dynamics NAV key.
- If another DataItem links to a given DataItem, then the reference field in the DataItemLink must be part of the same Microsoft Dynamics NAV key as the columns in the given DataItem.

Microsoft Dynamics NAV server automatically use a SIFT index for Query object whenever possible.

Differences Between Query and Record Result Sets

An important difference between Query Objects and Record Result Sets is that Microsoft Dynamics NAV does not do any caching for query result sets. When you run a query, Microsoft Dynamics NAV always issues a SQL select statement to get the data directly from SQL Server.

Also, Record result sets are always dynamic. For example, consider a situation where the user opens a Record result set which contains over 100 name and address records. While the user process is handling the 20th record it also inserts a new record which would qualify as the 80th record in the results set. Because Record result sets are always dynamic, when the user processes reaches the 80th record it will have access to the new record inserted earlier in the same process.

Query Object results sets are not guaranteed to be dynamic.

For further details, see the following topics in the Microsoft Dynamics NAV 2013 documentation:

- Queries
- SumIndexField Technology (SIFT)
- Optimizing SQL Server Performance with Microsoft Dynamics NAV

© 2012 Microsoft Corporation. All rights reserved.