MODULE 13: SQL SERVER OPTIMIZATION

Module Overview

Microsoft Dynamics NAV 2013 runs on Microsoft SQL Server. This module covers the integration between Microsoft Dynamics NAV with Microsoft SQL Server in more detail.

Objectives

The objectives are as follows:

- Explain the advantages of SQL Server for Microsoft Dynamics NAV 2013.
- Work with and store tables and indexes.
- Use collation options and descriptions.
- Introduce SQL Server Query Optimizer.
- Explain the areas within Microsoft Dynamics NAV that are to be optimized.
- Demonstrate how the Microsoft Dynamics® NAV database driver allows the Microsoft Dynamics NAV clients to communicate with SQL Server.
- Introduce the value of optimizing indexes to maximize performance.
- Describe the performance effect of locking, blocking, and deadlocks.
- Present how SIFT data is stored in SQL Server.

SQL Server for Microsoft Dynamics NAV

SQL Server is a comprehensive database platform that provides enterprise-class data management with integrated business intelligence (BI) tools. SQL Server can be characterized as a set-based engine. This means that SQL Server is very efficient when it retrieves a set of records from a table, but less so when records are accessed one at a time.

Access to SQL Server from Microsoft Dynamics NAV is performed with the Microsoft Dynamics database driver that is discussed later in this module. The SQL Server interface from Microsoft Dynamics NAV Server was rewritten for Microsoft Dynamics NAV 2013 to use ADO.NET instead of ODBC. The advantages of the new access layer are described later in this module.

When SQL Server receives a query (in the form of a Transact-SQL statement), it uses the SQL Query Optimizer to create and execute the query. The Query Optimizer evaluates the query and makes decisions about how to execute the query in the execution plan. For example, the Query Optimizer decides which index to use, whether to use parallel execution, and so on.

Query Optimizer assumes that the client generates queries according to its own logic, and that these queries are not optimized for SQL Server. The primary criteria that Query Optimizer uses to decide which execution plan to use is the performance cost of executing the query.

SQL Server stores data in B+ tree structures. One index is used to store the data physically on a disk. Other indexes are used to find a range and point to the data in the main index. This main index is called the *Clustered* Index. On SQL Server, you can define any index as the main (clustered) index. However, on the MicrosoftDynamics NAV Development Environment it is the default primary key of the table.

Note: *By default, the primary key of a table in Microsoft Dynamics NAV becomes the clustered index. You can use the key property Clustered to define another key to become the clustered index on SQL Server. We recommend that the primary key and the clustered index be the same.*

Microsoft Dynamics NAV 2013 no longer uses server cursors to retrieve records. Instead, records are retrieved by using multiple active result sets (MARS).

Representation of Microsoft Dynamics NAV Tables and Indexes in SQL Server

By default, Microsoft Dynamics NAV provides unique data for each company in its database. On SQL Server, each company in the development environment has its own copy of each table.

Representation of Microsoft Dynamics NAV Tables and Indexes in SQL Server

Each table in the Development Environment has a corresponding table in SQL Server for every company in the database, with a name in the following format:

Table Name Format	Example
<Company Name>$< Table Name>	CRONUS International Ltd_$G_L Entry

However, you can share data across companies by setting the DataPerCompany table property to FALSE. In Microsoft Dynamics NAV terms, this is known as *data common to all companies*. When the DataPerCompany property is turned off, there is just one table in SQL Server that is accessed from every company in the database. The naming convention for these common tables on SQL Server is the same, but without the <Company Name>$ portion.

The Microsoft Dynamics NAV Development Environment uses naming conventions that comply with SQL Server, such as not using special characters. Some special characters are available in the table designer, and they are translated to comply with the character set that is used on SQL Server.

The table has several indexes that represent the keys that are designed and enabled in the table designer. The indexes have generic names in the following format.

Index name format	Example
$<Index Number>	$1, $2, and so on

However, the primary key index uses the following name format.

Primary key name format	Example
<Company Name>$< Table Name>$0	CRONUS International Ltd_$G_L Entry$0

By default, Microsoft Dynamics NAV clusters the primary key. Also by default, Microsoft Dynamics NAV adds the rest of the primary key to every secondary index. This makes the indexes unique and complies with the best practices that are defined for SQL Server. Developers can make additional changes to the way indexes are defined on SQL Server by using the MaintainSQLIndex, SQLIndex, and Clustered properties on the keys that are defined in the table designer.

To obtain a list of indexes and their definition in SQL Server, run the sp_helpindex stored procedure in a query window, as follows.

Code Example

```
sp_helpindex "CRONUS International Ltd_$G_L Entry"
GO
```

The query outputs the index name if the index is clustered or unique, if there is a primary key constraint, and also the index keys that are defined in the index.

There are some differences between the Dynamics NAV and SQL Server terminology, as the following list describes.

SQL Server Terminology	Dynamics NAV Terminology
Primary key constraint	Primary key
Clustered index	No equivalent
Nonclustered index	Secondary key
Index key	Field in a key definition

In SQL Server, a table does not have to have a clustered index. This is known as a *heap* and can be used for archiving, because data is stored as it arrives. However, heaps are not ideal for tables that are read, because reading from an unstructured source is too slow.

Collation Options

SQL Server supports several collations. A *collation* encodes the rules that govern the correct use of characters for either a language, such as Macedonian or Polish, or an alphabet, such as Latin1_General (the Latin alphabet that is used by Western European languages). Microsoft Dynamics NAV 2013 only supports the latest Windows collations. Any database that is upgraded by Microsoft Dynamics NAV 2013 is converted to the most recent corresponding Windows collation. Each SQL Server collation specifies the following three properties:

- The sort order to use for Unicode data types (nchar, nvarchar, and ntext). A *sort order* defines the sequence in which characters are sorted, and the way that characters are evaluated in comparison operations.
- The sort order to use for non-Unicode character data types (char, varchar, and text).
- The code page that is used to store non-Unicode character data.

You can specify SQL Server collations at any level. Each instance of SQL Server has a defined default collation. This is the default collation for all objects in that instance of SQL Server, unless otherwise specified. Each database can have its own collation. This can differ from the default collation. You can specify separate collations for each column, variable, or parameter. Microsoft Dynamics NAV sets the database default collation for reference only. All columns that are created by Microsoft Dynamics NAV explicitly has the collation set.

It is a good practice to set the collation as generic as possible to the language that is most common to users. If all users speak the same language, set up SQL Server with a collation that supports that language. For example, if all users speak French, define a French collation on SQL Server. If users speak multiple languages, define a collation that best supports the requirements of the various languages. For example, if the majority of users speak western European languages, then Latin1_General collation is the best option.

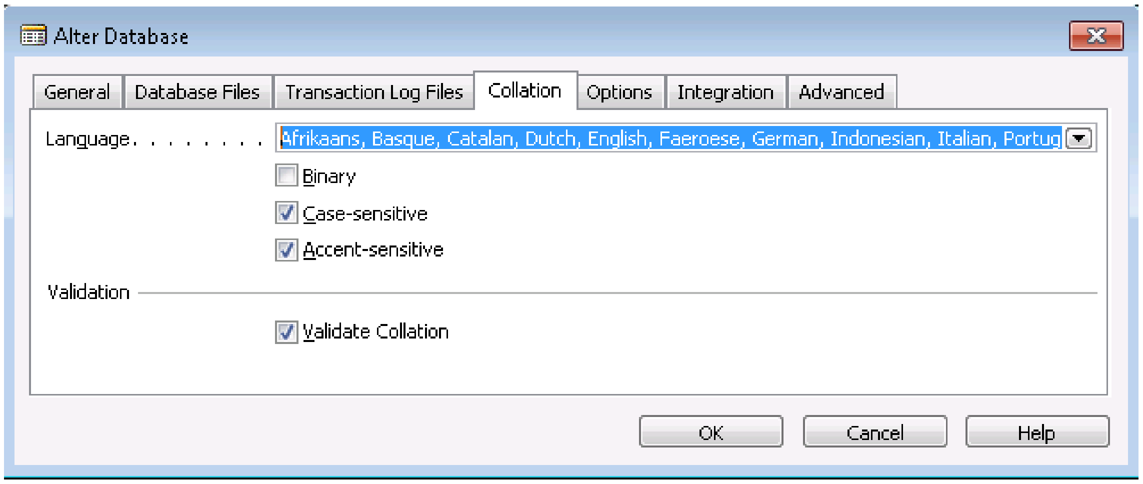
Collation settings are defined in the Microsoft Dynamics NAV Development Environment when a database is created. You can change it afterward with some limitations.

Demonstration: Open the Collation window

Ask Mort to verify the collation settings of the Dynamics NAV Database to update the documentation.

Demonstration Steps

1. To open the collation window follow these steps.
   a. Start the Microsoft Dynamics NAV Development Environment.
   b. In the **File** menu, click **Database** and then **Alter**. The **Alter Database** window opens.
   c. In the **Alter Database** window, click the **Collation** tab. See Collation Tab of the Alter Database Window.



Language: Afrikaans, Basque, Catalan, Dutch, English, Faeroese, German, Indonesian, Italian, Portug

[ ] Binary

[✓] Case-sensitive

[✓] Accent-sensitive

Validation

[✓] Validate Collation

FIGURE 13.1: THE COLLATION TAB OF THE ALTER DATABASE WINDOW

Collation Description

Select the name of a specific Windows collation from the drop-down list. Note that when the **Validate Code Page** field is checked, only valid subsets of collations are available in the list based on the Windows Locale. For example, the following are subsets for the Latin1_General (1252) locale consecutively:

- Afrikaans, Basque, Catalan, Dutch, English, Faeroese, German, Indonesian, Italian, Portuguese
- Danish, Norwegian

Sort Order

Select Sort Order options to use with the collation that you selected. These options are Binary, Case-sensitive, and Accent-sensitive. Binary is the fastest sorting order, and is case-sensitive and accent-sensitive. If you select Binary, the Case-sensitive and Accent-sensitive options are not available.

SQL Server Query Optimizer

Query Optimizer is the brain of SQL Server when it decides how to execute a query. SQL Server collects statistics about individual columns (single-column statistics) or sets of columns (multicolumn statistics). Query Optimizer uses statistics to estimate the selectivity of expressions, and therefore, the size of intermediate and final query results. Good statistics let the optimizer accurately assess the cost of different query plans and select a high-quality plan. All information about a single statistics object is stored in several columns of a single row in the sysindexes table, and in a statistics binary large object (statblob) that is kept in an internal-only table.

SQL Server Statistics

SQL Server maintains some information at the table level. Tables are not part of a statistics object, but SQL Server uses them occasionally during query cost estimation. The following data is stored at the table level:

- Number of rows in the table or index (rows column in sys.sysindexes)
- Number of pages that are occupied by the table or index (dpages column in sys.sysindexes)

SQL Server collects the following statistics about table columns and stores them in a statistics object (statblob):

- Time that the statistics are collected
- Number of rows that are used to produce the histogram and density information (described later)
- Average key length
- Single-column histogram that includes the number of steps

A *histogram* is a set of up to 200 values of a given column. All or a sample of the values in a given column are sorted. The ordered sequence is divided into up to 199 intervals so that the most statistically significant information is captured. Generally, these intervals are not of equal size.

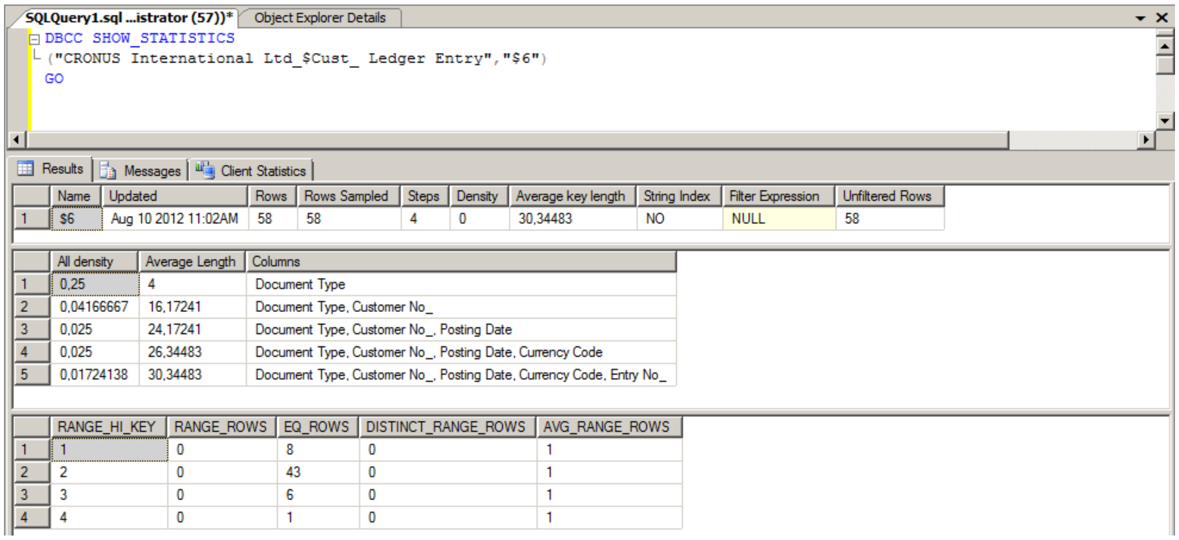
Note: *The way that Microsoft Dynamics NAV executes queries disables the use of histograms. You do this to enable performant reuse of query plans without regard to actual parameter values. Microsoft Dynamics NAV uses the OPTIMIZE FOR UNKNOWN option to guarantee this type of computation of the query execution plan.*

Users can view the statistical information when they run the DBCC SHOW_STATISTICS command. For example, they can run it for index $6 in the **Cust. Ledger Entry** table, as follows.

Code Example

```
DBCC SHOW_STATISTICS

("CRONUS International Ltd_$Cust_ Ledger Entry","$6")

GO
```

The result set has three sections, similar to the following.



```
DBCC SHOW_STATISTICS
("CRONUS International Ltd_$Cust_ Ledger Entry","$6")
GO
```

	Name	Updated	Rows	Rows Sampled	Steps	Density	Average key length	String Index	Filter Expression	Unfiltered Rows
1	$6	Aug 10 2012 11:02AM	58	58	4	0	30,34483	NO	NULL	58

	All density	Average Length	Columns
1	0,25	4	Document Type
2	0,04166667	16,17241	Document Type, Customer No_
3	0,025	24,17241	Document Type, Customer No_, Posting Date
4	0,025	26,34483	Document Type, Customer No_, Posting Date, Currency Code
5	0,01724138	30,34483	Document Type, Customer No_, Posting Date, Currency Code, Entry No_

	RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	1	0	8	0	1
2	2	0	43	0	1
3	3	0	6	0	1
4	4	0	1	0	1

FIGURE 13.2: DBCC SHOW_STATISTICS RESULTSET

Suppose that a user filters on the Document Type column in the **Cust. Ledger Entry** table, for example, to look for all Credit Memo type entries. The Credit Memo type entries have a value of Document Type that is equal to three. Microsoft Dynamics NAV issues a query similar to the following.

Code Example

```
SELECT * FROM

"CRONUS International Ltd_$Cust_ Ledger Entry"

WHERE

"Document Type" = 3

GO
```

The Query Optimizer analyzes the usefulness of every index in the table so that the query is executed at minimal cost. An index minimizes first the cost of data retrieval, followed by costs of sorting, and so on. When you analyze this particular index (index $6) from a data retrieval perspective, the Query Optimizer makes most of its decisions based on the statistics in the following way.

Document Type is filtered because there are only three distinct values in the index (refer to the All Density column in the previous table). This indicates that 0.25 of the table is within the filtered set.

Based on this, the Query Optimizer decides that there is no point in using this index to do this operation. This is because it has to load the index, scan the range, and look up the data in the clustered indexes to return the results. Because there is no better way to read the data, Query Optimizer decides to do a Clustered Index Scan instead.

As a rule, if the selectivity is close and better than one percent, the index is considered good. Be careful with this simple rule, because operations such as SELECT TOP 1 (asking for the first record in a set) escalate the index benefit, and the index will probably be used.

To continue with this example, filtering on a unique value makes the index help with selectivity, such as the following.

Code Example

```
SELECT * FROM

"CRONUS International Ltd_$Cust_ Ledger Entry"

WHERE

"Document Type" = 3 AND

"Customer No_" = '10000'

GO
```

The combined selectivity is used, and a plan is calculated. This may result in a decision to use this index.

However, if users do not filter on an index key or use the <> (not equal operator) or use the OR operator, SQL Server cannot combine the subqueries. If the <> operator is used, then the optimizer will assume 1-selectivity * number of rows. If OR is used, the optimizer combines the selectivity. This may result in bad behavior, such as the following.

Code Example

```
SELECT * FROM

"CRONUS International Ltd_$Cust_ Ledger Entry"

WHERE

"Customer No_" = '10000' AND
```

```
(("Document Type" = 3) OR ("Document Type" = 4))

GO
```

In this example, the Query Optimizer decides to use the index, doing a non-clustered index seek, but it has to traverse the area of Document Type = 3. Because the set is read from start to finish, it has the same effect as if the user did a table scan.

Similarly, if the user leaves one of the index keys unfiltered, all the subtree index entries must be scanned. There is one simple rule about performance: Scan is bad. Seek is good. Developers must avoid scans as much as they can, ensure that indexes are of a good selectivity, and that the queries do not have scan-like behavior.

On the other side of the spectrum is an index that is too complex and is designed to fully match the whole query, such as with eight index keys. If the index has only four index keys, SQL Server scans a slightly larger set to provide the required records, but at the extreme cost of having to maintain the composite index. This delays every modification in the table, because SQL Server has to update the index accordingly. In most cases, users have to optimize the transaction speed. If you over-index the tables, then users pay a price in performance. If a specific report is slow when fewer composite indexes are used, it might be worth it because processing (such as posting inventory) is quicker.

Additionally, if a fairly small set is ordered by a different column, it is not necessary for the index to fully support the sorting. SQL Server can efficiently sort small result sets quickly.

The previous situation demonstrates that the way that indexes are designed and used can severely affect SQL Server performance. Use the following principles:

- Reduce the number of indexes for faster table updates.
- Design indexes with index keys of good selectivity.
- Put index keys that are more likely to be filtered toward the beginning of an index.
- If the filtered index keys point to a low number of records, you do not have to add additional index keys to support index selectivity or sorting. SQL Server returns the set sorted as you want.
- There is no point in indexing empty (unused) columns, or columns that have the same value for all rows. This creates an additional overhead with no benefit.
- Make sure that users filter on unique values in indexes; otherwise, SQL Server performs in a manner that is similar to table scans.

- Do not make *over selective* index keys. If users index on **DateTime** fields, for example, they force creation of a unique index leaf in the index for each record in the table.
- Put date fields toward the end of the index, since the index is not always filtered. If an index is always filtered on a unique value, then it is a good index.

To determine whether an index is good, imagine a telephone book or list of personal details that are designed to find people by name and surname, date of birth, or Social Security number. Compare this to a telephone book that is indexed by gender. When you design indexes, select those that support a high level of selectivity. Common sense applies.

Optimize a Microsoft Dynamics NAV Application

There are several areas where users must focus when they optimize Microsoft Dynamics NAV applications. These areas follow, in order of importance (based on the processing costs):

- SIFT
- Indexes
- Locks
- Suboptimum code
- GUI

Optimizing SIFT Tables

Use SIFT tables in Microsoft Dynamics NAV version 5.0 and older, to implement SIFT on the SQL Server, and store aggregate values for SumIndexFields for keys in the source tables. Starting with version 5.0 Service Pack 1, these SIFT tables are replaced by indexed views. Separate SIFT tables are no longer part of Microsoft Dynamics NAV on SQL Server. This discussion section is included because Microsoft Dynamics NAV developers can encounter issues about SIFT tables in implementations of older versions of Microsoft Dynamics NAV. There is similar overhead with indexed views.

The overhead of the separate SIFT indexes is very large and should be carefully considered for activation. By default, Microsoft Dynamics NAV starts the SIFT indexes when users create a new index with SumIndexFields. Users should review all existing SIFT indexes and decide whether they really have to keep them started.

Users can deactivate the creation and maintenance of a SIFT index by using the MaintainSIFTIndex property in the Microsoft Dynamics NAV key designer. If they make the property FALSE, and there is no other maintained SIFT index that supports the retrieval of the cumulative sum, Microsoft Dynamics NAV asks SQL Server to calculate the total.

For example, if users have a **Sales Line** table and put Amount in the SumIndexFields for the primary key (Document Type, Document No., Line No.), a new SIFT table named **CRONUS International Ltd_$37$0** is created and maintained. When you use a **CALCSUM** function to display a FlowField in Microsoft Dynamics NAV that displays the total of all Sales Lines for a specific Sales Header (Order ORD-980001), the resulting query looks exactly like the following.

Code Example

```
SELECT SUM(SUM$Amount) FROM

"CRONUS International Ltd_$Sales Line$VSIFT$0"

WHERE

"Document Type" = 1 AND

"Document No_" = 'ORD-980001'
```

If users make the SIFT table unavailable by clearing the **MaintainSIFTIndex** check box, Microsoft Dynamics NAV still works, and the resulting query resembles the following.

Code Example

```
SELECT SUM("Amount") FROM

"CRONUS International Ltd_$Sales Line"

WHERE

"Document Type" = 1 AND

"Document No_" = 'ORD-980001'
```

This is a very light load on CPU overhead compared to the large cost of maintaining the SIFT indexes.

SIFT tables are very useful when users have to total a larger number of records. Considering this information, users can check existing SIFT indexes and see whether they need some level of detail. There is no need, for example, to store a cumulative total of just a few records.

Optimize Indexes

The second largest Microsoft Dynamics NAV overhead is the processing load to maintain indexes. The Microsoft Dynamics NAV database is over-indexed, because customers require certain reports and pages to be ordered in different ways. Therefore, many keys are made available for each sequence. However, SQL Server can sort data directly and quickly if the set is small. Therefore, you do not have to keep indexes for sorting purposes only. For example, in the **Warehouse Activity Line** table, there are several keys that begin with **Activity Type** and **No.** fields, such as the following.

Code Example

```
“Activity Type,No.,Sorting Sequence No.”
“Activity Type,No.,Shelf No.”
“Activity Type,No.,Action Type,Bin Code”
etc.
```

These indexes are not needed on SQL Server, because the Microsoft Dynamics NAV code always filters on **Activity Type** and **No.** fields when it uses these keys. In SQL Server, the Query Optimizer looks at the filter and realizes that the clustered index is Activity Type No_, and Line No_. It also determines that the set is small, and that it does not have to use an index to retrieve the set and return it in that specific order. It uses only the clustered index for these operations.

Since the whole functionality is not used by customers, if they never select the stock by **Sorting Sequence No.**, then they do not have to maintain the index.

Developers must analyze the existing indexes and focus on use and benefits compared to the processing overhead, and then determine the appropriate action. Decide between disabling the index completely by using the key property Enable, the KeyGroups property, or the MaintainSQLIndex property. Indexes that remain active can change structure by using the SQLIndex property. Developers can also make the table clustered by a different index.

Enabled Property The Enabled property turns a specific key on and off. If you are not using the key or if you rarely use the key, you may want to mark it as disabled for performance reasons.

KeyGroups Property

Make one or more keys a member of a predefined key group. This allows for the key to be defined, but enabled only when it is used. Use the KeyGroups property to select the predefined key groups. Select the KeyGroups option on the **Database Information** window (select **File > Database > Information > Tables**). There are key groups already defined, such as Acc(Dim), Item(MFG), but more can be created and assigned to keys.

The purpose of key Key groups set up a group of special keys that are infrequently used (such as for a special report that is run one time every year). Adding many keys to tables eventually decreases performance. When you use key groups it makes it possible to have the necessary keys defined, but only active when it is necessary.

MaintainSQLIndex Property

This property determines whether an SQL Server index that corresponds to the Microsoft Dynamics NAV key should be created (when set to **Yes**) or dropped (when set to **No**). A Microsoft Dynamics NAV key is created to sort and search for data in a table by the required key fields. However, SQL Server can sort data without an index on the fields to be sorted. If an index exists, sorting by the fields matching the index is faster, but modifications to the table will be slower. The more indexes there are on a table, the slower the modifications become. If a key must be created to allow for only occasional sorting (for example, when it is running infrequent reports), users can disable this property to prevent slow modifications to the table. Additionally, if there are many keys in a table, SQL Server does not use all the corresponding indexes. To eliminate the overhead of the indexes we recommend not maintaining them by setting MaintainSQLIndex to **No**. Then, if there is a reference in the C/AL code to the key, it does not cause an error message because the key still exists.

SQLIndex Property

This property lets users define the fields that are used in the SQL index. The fields in the SQL index can be any of the following:

- Different from the fields that are defined in the key in Microsoft Dynamics NAV. There can be fewer fields or more fields.
- Arranged in a different order.

If the key in question is not the primary key and you use the SQLIndex property to define the index on SQL Server, the index that is created contains exactly the fields that users specify. It is not necessarily a unique index. It will only be a unique index if it contains all fields from the primary key.

If you define the SQL index for the primary key, it must include all the fields that are defined in the Microsoft Dynamics NAV primary key. You can add additional fields that can be rearranged to suit individual needs.

Clustered Property

Use this property to determine which index is clustered. By default, the index that corresponds to the Microsoft Dynamics NAV primary key is clustered.

The Index Usage Query

The Index Usage Query was released on the Microsoft Dynamics NAV Team Blog and can be found in the following location.

Index Usage Query on Dynamics NAV Team Blog

http://go.microsoft.com/fwlink/?LinkId=269777

This query shows a list of all tables and indexes in an SQL database to help identify tables where the most blocks occur, and to give a starting point for index tuning.

Use the Index Information Query to see the number of records in each table. By changing ORDER BY, you also can use it to see which index causes the most blocking, wait time, updates, or locks. Use the Index Information Query to compare Index Updates with Index Reads for an idea of cost versus benefit for each index. Then you can use this information to decide whether to maintain certain indexes.

To summarize, the Index Information Query provides information about the following:

- Index and Table Information
- Index usage (benefits and costs information for each index)
- Index locks, blocks, wait time, and updates per read (cost versus benefit)

You must execute the query in the Microsoft Dynamics NAV database on SQL Server. It creates a table named **z_IUQ2_Temp_Index_Keys** and uses various Microsoft Dynamic Management Views to collect information for each index for this table. When you execute the complete query, it can take several minutes to execute for each company in the database. If you just want to change sorting or display the results again later, you only have to run the last part of the query as follows.

Code Example

```
-- Select results

SELECT

[F_Table_Name] TableName,

[F_Row_Count] No_Of_Records,

[F_Data] Data_Size,
[F_Index_Size] Index_Size,

[F_Index_Name] Index_Name,

[F_User_Updates] Index_Updates,

[F_User_Reads] Index_Reads,

CASE WHEN

F_User_Reads = 0 THEN F_User_Updates

ELSE

F_User_Updates / F_User_Reads

END AS Updates_Per_Read,

[F_Locks] Locks,

[F_Blocks] Blocks,

[F_Block_Wait_Time] Block_Wait_Time,

[F_Last_Used] Index_Last_Used,

[F_Index_Type] Index_Type,

[Index_Key_List] Index_Fields

FROM z_IUQ2_Temp_Index_Keys

--order by F_Row_Count desc, F_Table_Name, [F_Index_ID]

--order by F_User_Updates desc

--order by Blocks desc

--order by Block_Wait_Time desc

--order by Updates_Per_Read desc

ORDER BY F_Table_Name
```

The last lines suggest various other ORDER BY clauses that you can use to replace the default ORDER BY clause that is by Table Name. There are several other columns that are available. You can easily change the query, for example ORDER BY user updates, to see the indexes that are causing the largest overheads, and then check the actual usage of these indexes.

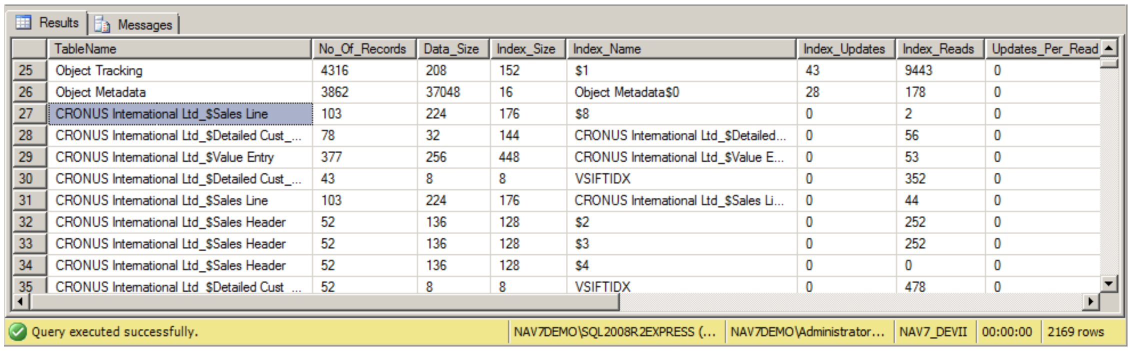
Code Example

```
--order by F_Row_Count desc, F_Table_Name, [F_Index_ID]

--order by F_User_Updates desc

--order by Blocks desc

--order by Block_Wait_Time desc

--order by Updates_Per_Read desc

ORDER BY F_Table_Name
```

To analyze query results, it is important to understand that the Index Information Query uses SQL Server Dynamic Management Views to collect index information. Some of these Dynamic Management Views access their information from the SQL Server cache. The cache is located in the RAM memory of the server. If you recently restarted SQL Server, then enough time may not have elapsed to warm up the cache. This means that the cache is not representing the actual workload on the server. Therefore, the results of the Index Information Query are misleading. To avoid this scenario, make sure that before you run the Index Information Query, enough time has elapsed since the last restart of SQL Server.

When the query results contain NULL values for most of the indexes in the database, it means that there is not enough information in the cache about the index usage. To obtain solid results, wait until a representative workload processes in Microsoft Dynamics NAV.

The following is an example of the query results on a Microsoft Dynamics NAV Demo Database.



	TableName	No_Of_Records	Data_Size	Index_Size	Index_Name	Index_Updates	Index_Reads	Updates_Per_Read
25	Object Tracking	4316	208	152	$1	43	9443	0
26	Object Metadata	3862	37048	16	Object Metadata$0	28	178	0
27	CRONUS International Ltd_$Sales Line	103	224	176	$8	0	2	0
28	CRONUS International Ltd_$Detailed Cust_...	78	32	144	CRONUS International Ltd_$Detailed...	0	56	0
29	CRONUS International Ltd_$Value Entry	377	256	448	CRONUS International Ltd_$Value E...	0	53	0
30	CRONUS International Ltd_$Detailed Cust_...	43	8	8	VSIFTIDX	0	352	0
31	CRONUS International Ltd_$Sales Line	103	224	176	CRONUS International Ltd_$Sales Li...	0	44	0
32	CRONUS International Ltd_$Sales Header	52	136	128	$2	0	252	0
33	CRONUS International Ltd_$Sales Header	52	136	128	$3	0	252	0
34	CRONUS International Ltd_$Sales Header	52	136	128	$4	0	0	0
35	CRONUS International Ltd_$Detailed Cust_...	52	8	8	VSIFTIDX	0	478	0

Query executed successfully. | NAV7DEMO\SQL2008R2EXPRESS (... | NAV7DEMO\Administrator... | NAV7_DEVII | 00:00:00 | 2169 rows

FIGURE 13.3: RESULTS OF THE INDEX USAGE QUERY

The column on the left side shows data for the table (No. of records, data and index size) where you can view the effect of indexes on the table.

The columns on the right side show data for each index. This includes Updates (costs) and Reads (benefits), and when it was last used since the last time SQL Server was restarted.

The following table describes the different columns of the Index Usage Query.

Field Name	Description
TableName	The name of the table.
No_Of_Records	Number of records in the table.
Data_Size	The current size of table data.
Index_Size	The current size of the index.
Index_Name	The name of the index.
Index_Updates	The number of times the index was updated by SQL Server.
Index_Reads	The number of times the index was read by SQL Server.
Updates_Per_Read	The number of Updates divided by the number of reads.
Locks	The number of times the index was involved in a lock.
Blocks	The number of times the index was involved in a block.
Block_Wait_Time	The time that the index was blocked (in ms).
Index_Last_Used	The datetime the index was last used.
Index_Type	The type of index (Clustered, NonClustered.)
Index_Fields	The fields of the index.

Define Keys to Improve Performance

When you write C/AL code that searches through a subset of the records in a table, you must consider the keys that are defined for the table and then write code that optimizes for the keys. For example, the entries for a specific customer are usually a small subset of a table that contains entries for all customers.

The time that is required to complete a loop through a subset of records depends on the size of the subset. If a subset cannot be located and read efficiently, then performance deteriorates.

To maximize performance, you must define the keys in the table that support the code that you run. Then you must specify these keys correctly in your code.

For example, to retrieve the entries for a specific customer, you apply a filter to the **Customer No.** field in the **Cust. Ledger Entry** table. To run the code efficiently on Microsoft SQL Server, you must define a key in the table that has **Customer No.** as the first field.

The table could have the following keys:

- Entry No.
- Customer No.
- Posting Date

The following is an example of code that finds a subset of records.

Code Example

```
SETRANGE("Customer No.",'1000');

IF FIND('-') THEN

REPEAT

UNTIL NEXT = 0;
```

SQL Server automatically selects the index to use to retrieve data in the most efficient way. SQL Server calculates the cost of retrieving data by using different indexes. Then it selects the path that has the smallest cost. For Microsoft Dynamics NAV, that calculation is based only on the statistical distribution of values in a column.

For example, if a table contains 1000 rows and a column in the table contains either the value 0 or the value 1, then that column is said to have a *low selectivity*. If a column contains the values ranging from 1 to 500, then the column is said to have a *high selectivity*. In the following code example, SQL Server selects an index that contains the HighSelectivityColumn. Then it sorts the rows by the LowSelectivityColumn.

Code Example

```
SETCURRENTKEY("LowSelectivityColumn");

SETFILTER("LowSelectivityColumn",'1');

SETFILTER("HighSelectivityColumn",'777');

FIND('-')
```

Implicit/Explicit Locking

There are additional considerations to make when you work with Microsoft Dynamics NAV on SQL Server. Microsoft Dynamics NAV is designed to read without locks, and it locks only if it is necessary. If records will be changed, indicate that intent (use explicit locking) so that the data is read correctly.

Implicit Locking

The following table demonstrates implicit locking. The C/AL pseudo-code on the left is mapped to the equivalent action on SQL Server.

Sample code	Result

TableX.FIND('-');	SELECT * FROM TableX
	  WITH (READUNCOMMITTED)
	
	(the retrieved record time stamp = TS1)

TableX.Field1 := Value;	

TableX.MODIFY;	performs the update
	UPDATE TableX
	  SET Field1 = Value
	  WHERE TimeStamp <= TS1

The call to TableX.MODIFY will implicitly lock the table from that point forward. This means that any call to FIND made on the table after this point will have the WITH(UPDLOCK) applied.

Because the record was read without locking the record, NAV guarantees data consistency by automatically adding a check in the where clause on the timestamp. This means that other users' changes are not overwritten.

Explicit Locking

If users indicate that they plan to modify the record by using explicit locking, then the SQL sent is slightly different, as shown by the following pseudo code.

Sample code	Result
TableX.LOCKTABLE;	Indicates to explicit lock.
TableX.FIND('-');	SELECT * FROM TableX WITH (UPDLOCK) (the retrieved record timestamp = TS1)
TableX.Field1 := Value;	

TableX.MODIFY;	UPDATE TableX
	SET Field1 = Value
	(The retrieved record timestamp is guaranteed to be TS1.)

Problems with NEXT

Sometimes, the NEXT command causes performance problems in Microsoft Dynamics NAV. Users should pay particular attention and avoid these situations. The problem is if users change the content or the definition of the result-set, then the set must be retrieved again. Microsoft Dynamics NAV guarantees that all changes that are made by the current user are included in a result-set. This is known as a *dynamic result-set*.

Be aware that if during traversal of a result-set any of following actions are performed, then retrieval of the result-set will be reset (Issue a new SQL statement):

- Changed filter
- Changed sorting
- Modified key value.
- Inserted a record.
- Modified, deleted, or inserted a record by using another instance of a record.
- Changed transaction type.

The following code examples demonstrate this problem.

Code	Result
SETCURRENTKEY(FieldA);	
SETRANGE(FieldA,Value);	
FIND('-')	Set or part of the set is retrieved, first record loaded.
REPEAT	
FieldA := NewValue;	Record position is now outside the set.
MODIFY;	Record is put outside the set.
UNTIL NEXT = 0;	The system is asked to go to NEXT from position, but from outside the set.

"Jumping" through data - NEXT "from current record content

Code	Result
SETCURRENTKEY(FieldA);	
SETRANGE(FieldA,Value);	
FIND('-');	Set based on filter on FieldA.
REPEAT	
…	
…	
FieldA := <SomeValue>	The current key value was changed. So, the call to next has to retrieve the record conforming with being greater than <SomeValue>.
…	
UNTIL NEXT = 0	The system is asked go to NEXT from undefined position.

FindFirst(only) followed by call to next

Code	Result
SETRANGE(FieldA,Value);	
FINDFIRST;	This call is optimized for retrieving the first record only. Therefore, calling next after this issued a new SQL query.
REPEAT	
…	
UNTIL NEXT = 0;	The system is asked to go to NEXT on non-existing set.
SETRANGE(FieldA,Value);	

Solutions

To eliminate performance problems with NEXT, consider the following solutions:

- Use a separate looping variable.
- Read records to temporary tables, modify within, and write back afterwards.

Suboptimum Coding and Other Performance Penalties

Taking performance into consideration frequently influences programming decisions. For example, if users do not use explicit locking, or if the program loops and provokes problems with NEXT, they frequently pay a big price in performance. Users also have to review their code and see how many times the code reads the same table, or use ISEMPTY or COUNT for checking if there is a record (IF COUNT = 0, IF COUNT = 1. Additionally, there are features in the application that must be avoided or minimized. Users should review the application setup for the performance aspect and take corrective actions if they can.

Data Access Redesign

The SQL Server interface from Microsoft Dynamics NAV Server was rewritten for Microsoft Dynamics NAV 2013 to use ADO.NET instead of ODBC.

Simplified Deployment

The new ADO.NET interface is a managed data access layer that supports SQL Server connection pooling. This can significantly decrease memory consumption by Microsoft Dynamics NAV Server. SQL Server connection pooling also simplifies deployment of the Microsoft Dynamics NAV three-tier architecture for deployments where the three tiers are installed on separate computers. Specifically, administrators are no longer required to manually create SPNs or to set up delegation when the client, Microsoft Dynamics NAV Server, and SQL Server are on separate computers.

Decreased Resource Consumption

There is no longer a one-to-one correlation between the number of client connections and the number of SQL Server connections. In earlier versions of Microsoft Dynamics NAV, each SQL Server connection could consume up to 40 MB of memory. Memory allocation is now in managed memory. This is generally more efficient than unmanaged memory.

In Microsoft Dynamics NAV 2013, all users who are connected to the same Microsoft Dynamics NAV Server instance share the data cache. This means that after one user has read a record, a second user who reads the same record retrieves it from the cache. In earlier versions of Microsoft Dynamics NAV, the data cache was isolated for each user.

Caching

Microsoft Dynamics NAV 2013 uses an improved cache system. The following functions use the cache system:

- GET
- FIND
- FINDFIRST
- FINDLAST
- FINDSET
- COUNT
- ISEMPTY
- CALCFIELDS
- CALCSUMS

There are two types of caches:

- Global cache – For all users who are connected to a Microsoft Dynamics NAV Server instance.
- Private cache – For each user, for each company, in a transactional scope. Data in a private cache for a given table and company are flushed when a transaction ends.

The lock state of a table determines the cache to use. If a table is not locked, then the global cache is queried for data. Otherwise, the private cache is queried.

Results from query objects are not cached.

For a call to any of the FIND functions, 1024 rows are cached. You can set the size of the cache by using the Data Cache Size setting in the Microsoft Dynamics NAV Server configuration file. The default size is 9. This approximates a cache size of 500 MB. If you increase this number by one, then the cache size doubles.

You can bypass the cache by using the **SELECTLATESTVERSION** Function.

Microsoft Dynamics NAV 2013 synchronizes caching between Microsoft Dynamics NAV Server instances that are connected to the same database. By default, synchronization occurs every 30 seconds.

You can set the cache synchronization interval by using the CacheSynchronizationPeriod parameter in the CustomSettings.config file.

Improved Performance

Microsoft Dynamics NAV 2013 no longer uses server cursors to retrieve records. Instead, you retrieve records by using multiple active result sets (MARS). Functions such as **Next**, **Find('-')**, **Find('+')**, **Find('>')**, and **Find('<')** are generally faster with MARS.

Note: *Because Microsoft Dynamics NAV 2013 no longer uses server cursors to retrieve records, the Record Set property under Caching on the **Advanced** tab of the **Alter Database** page was no longer needed and was removed.*

SIFT indexes also are improved. For example, COUNT and AVERAGE formulas can now use SIFT indexes. MIN and MAX formulas now use SQL Server MIN and MAX functions exclusively.

RecordId's and SQL Variant columns in a table no longer prevent use of BULK insert inserts.

In most cases, filtering on FlowFields issues a single SQL statement. In earlier versions of Microsoft Dynamics NAV, filtering on FlowFields issued an SQL statement for each filtered FlowField and for each record in the table to calculate the filtered FlowFields. The following exceptions are in Microsoft Dynamics NAV 2013 in which filtering on FlowFields does not issue a single SQL statement:

- You use the ValueIsFilter option on a field and the field has a value.
- A second predicate is specified on a source field and the field that is used for the second predicate has a value. For example, when you specify the CalcFormula Property for a FlowField, you can specify table filters in the **Calculation Formula** window. If you specify two or more filters on the same source field, then filtering does not issue a single SQL statement.
- You specify Validated for the SecurityFiltering Property on a record. This value for the SecurityFiltering property means that each record that is part of the calculation must be verified for inclusion in the security filter.

In most cases, calling the **FIND** or **NEXT** functions after you set the view to include only marked records issues a single SQL statement. In earlier versions of Microsoft Dynamics NAV, calling **FIND** or **NEXT** functions that have marked records issued an SQL statement for each mark. There are some exceptions if many records are marked.

C/AL Database Functions and Performance on SQL Server

GET, FIND, and NEXT

The C/AL language offers several methods to retrieve record data. Each of the following functions is optimized for a specific purpose. To achieve optimal performance you must use the method that is best suited for a given purpose.

- **Record.GET –** This function is optimized for retrieving a single record based on primary key values.
- **Record.FIND –** The **FIND** function is optimized for retrieving a single record based on the primary keys in the record and any filter or range that was set.
- **Record.FIND('-') and Record.FIND('+') –** These functions are optimized for reading an open-end dataset when the application might not read all records.
- **Record.FINDSET(ForUpdate, UpdateKey) –** The **FINDSET** function is optimized for reading the whole set of records within the specified filter and range. The UpdateKey parameter does not influence the efficiency of this method in Microsoft Dynamics NAV 2013.

- **Record.FINDFIRST and Record.FINDLAST –** The **FINDFIRST** and **FINDLAST** functions are optimized for finding the single first or last record within the specified filter and range. Use these functions when you only require one record.
- **Record.FINDFIRST and Record.FINDLAST –** The **FINDFIRST** and **FINDLAST** functions are optimized for finding the single first or last record within the specified filter and range. Use these functions when you only require one record.
- **Record.FINDFIRST and Record.FINDLAST –** The **FINDFIRST** and **FINDLAST** functions are optimized for finding the single first or last record within the specified filter and range. Use these functions when you only require one record.
- **Record.FINDFIRST and Record.FINDLAST –** The **FINDFIRST** and **FINDLAST** functions are optimized for finding the single first or last record within the specified filter and range. Use these functions when you only require one record.
- **Record.NEXT –** The NEXT function can be called at any time. However, if Record.NEXT is not called as part of retrieving a continuous result set, then Microsoft Dynamics NAV calls a separate SQL statement in order to find the next record.

Dynamic Result Sets

In Microsoft Dynamics NAV any result set that is returned from a call to the find methods discussed in the previous section is dynamic. This means that the result set is guaranteed to contain any changes that you make later in the result set. However this feature comes at a cost. If any modifications are made to a table that is being traversed, then Microsoft Dynamics NAV might have to issue an additional SQL statement to guarantee that the result set is dynamic.

The following code shows how records are most efficiently retrieved. FINDSET is the most efficient method to use because it reads all records.

Code Example

```
IF FINDSET THEN

REPEAT

  // Insert statements to repeat.

UNTIL NEXT = 0;
```

CALCFIELDS, CALCSUMS, and COUNT Functions

Each call to **CALCFIELDS**, **CALCFIELD**, **CALCSUMS**, or **CALCSUM** functions that calculates a sum requires a separate SQL statement. This is true unless the client calculated the same sum or another sum that uses the same SumIndexFields or filters in a recent operation.

Each **CALCFIELDS** or **CALCSUMS** function request should be confined to use only one SIFT index. You can use the SIFT index only as follows:

- All requested sum-fields are contained in the same SIFT index.
- The filtered fields are part of the key fields that are specified in the SIFT index that contains all the sum fields.

If neither of these requirements is fulfilled, then the sum is calculated directly from the base table.

In Microsoft Dynamics NAV 2013, use SIFT indexes to count records in a filter, as long as a SIFT index exists that contains all filtered fields in the key fields that are defined for the SIFT index.

SETAUTOCALCFIELDS

It is a common task to retrieve data and request calculation of associated FlowFields. The following example traverses customer records, calculates the balance, and marks the customer as blocked if the customer exceeds the maximum credit limit.

Code Example

```
IF Customer.FINDSET() THEN REPEAT

  Customer.CALCFIELDSS(Customer.Balance)

  IF (Customer.Balance > MaxCreditLimit) THEN BEGIN

    Customer.Blocked = True;

    Customer.MODIFY();

  END

  ELSE IF (Customer.Balance > LargeCredit) THEN BEGIN

    Customer.Caution = True;

    Customer.MODIFY();
  END;

UNTIL Customer.NEXT = 0;
```

In Microsoft Dynamics NAV 2013, you can do this much faster. First, set a filter on the customer. You can also do this in Microsoft Dynamics NAV 2009, but behind the scenes the same code is executed. In Microsoft Dynamics NAV 2013, setting a filter on a record translates into a single SQL statement.

Code Example

```
Customer.SETFILTER(Customer.Balance,’>%1’, LargeCredit);

IF Customer.FINDSET() THEN REPEAT

  Customer.CALCFIELDS(Customer.Balance)

  IF (Customer.Balance > MaxCreditLimit) THEN BEGIN

    Customer.Blocked = True;

    Customer.MODIFY();

  END

  ELSE IF (Customer.Balance > LargeCredit) THEN BEGIN

    Customer.Caution = True;

    Customer.MODIFY();

  END;

UNTIL Customer.NEXT = 0;
```

In the previous example, an additional call to the CALCFIELDS function still must be issued for the code to check the value of Customer.Balance. You can optimize this more by using the new **SETAUTOCALCFIELDS** function.

Code Example

```
Customer.SETFILTER(Customer.Balance,’>%1’, LargeCredit);

Customer.SETAUTOCALCFIELDS(Customer.Balance)

IF Customer.FINDSET() THEN REPEAT

  IF (Customer.Balance > MaxCreditLimit) THEN BEGIN
```

```
   Customer.Blocked = True;

   Customer.MODIFY();

  END

  ELSE IF (Customer.Balance > LargeCredit) THEN BEGIN

   Customer.Caution = True;

   Customer.MODIFY();

  END;

UNTIL Customer.NEXT = 0;
```

INSERT, MODIFY, DELETE, and LOCKTABLE

Each call to **INSERT**, **MODIFY**, or **DELETE** functions requires a separate SQL statement. If the table that you modify contains SumIndexes, then the operations are significantly slower. As a test, select a table that contains SumIndexes and execute one hundred **INSERT**, **MODIFY**, or **DELETE** operations to measure how long it takes to maintain the table and all its SumIndexes.

The **LOCKTABLE** function does not require any separate SQL statements. It only causes any successive reading from the table to lock the table or parts of it.

Bulk Inserts

Microsoft Dynamics NAV automatically buffers inserts to send them to Microsoft SQL Server at one time.

By using bulk inserts, the number of server calls is reduced. This improves performance.

Bulk inserts also improve scalability by delaying the actual insert until the last possible moment in the transaction. This reduces the time that tables are locked, especially tables that contain SIFT indexes.

Software developers who want to write high performance code that uses this feature should understand the following bulk insert constraints.

Bulk Insert Constraints

If you want to write code that uses the bulk insert functionality, you must be aware of the following constraints.

Records are sent to SQL Server when the following occurs:

- You call COMMIT to commit the transaction.
- You call MODIFY or DELETE on the table.
- You call any FIND or CALC methods on the table.
- Records are not buffered if one of the following conditions is TRUE:
- The application is using the return value from an INSERT call, for example, "IF (GLEntry.INSERT) THEN".
    - The table that you insert the records into contains any of the following:
        - BLOB fields
        - Fields that have the AutoIncrement property set to Yes

The following code example cannot use buffered inserts because it contains a FIND call on the **GL/Entry** table within the loop.

Code Example

```
IF (JnlLine.FIND('-')) THEN BEGIN

  GLEntry.LOCKTABLE;

  REPEAT

    IF (GLEntry.FINDLAST) THEN

      GLEntry.NEXT := GLEntry."Entry No." + 1

    ELSE

      GLEntry.NEXT := 1;

    // The FIND call will flush the buffered records.

    GLEntry."Entry No." := GLEntry.NEXT ;

    GLEntry.INSERT;

  UNTIL (JnlLine.FIND('>') = 0)

END;

COMMIT;
```

If you rewrite the code as shown in the following example, you can use buffered inserts.

Code Example

```
IF (JnlLine.FIND('-')) THEN BEGIN

  GLEntry.LOCKTABLE;

  IF (GLEntry.FINDLAST) THEN

    GLEntry.Next := GLEntry."Entry No." + 1

  ELSE

    GLEntry.Next := 1;

  REPEAT

    GLEntry."Entry No.":= GLEntry.Next;

    GLEntry.Next := GLEntry."Entry No." + 1;

    GLEntry.INSERT;

  UNTIL (JnlLine.FIND('>') = 0)

END;

COMMIT;

// The inserts are performed here.
```

Locking, Blocking, and Deadlocks

When data is read from the database, Microsoft Dynamics NAV uses the READUNCOMMITTED isolation level. This means that any user can modify the records that are currently being read, until the table is either changed by a write operation, or locked with the **Record.LOCKTABLE** function. From this point until the end of the transaction, all read operations on the table are performed with both REPEATABLE READ and UPDLOCK locking. This is frequently known as *pessimistic concurrency*.

Locking

Records can be read with different types of locks such as UPDLOCK. At this level, records that are read are locked. This means that no other user can modify the record.

An example of a lock of a customer record can be demonstrated by the following code.

Code Example

```
Customer.LOCKTABLE;

Customer.GET('10000');        // Customer 10000 is locked

Customer.Blocked := Customer.Blocked::All;

Customer.MODIFY;

COMMIT;                       // Lock is removed
```

If the record is not locked, the following situation can occur:

User A	User B	Comment
Customer.GET('10000') ;		User A reads record without any lock.
	Customer.GET('10000') ;	User B reads same record without any lock.
...	Customer.Blocked := Customer.Blocked::All ; Customer.MODIFY; COMMIT;	User B modifies record.
Customer.Blocked := Customer.Blocked::""; Customer.MODIFY;		User A gets an error: "Another user has modified the record...".
ERROR	SUCCESS	

Blocking

When other users try to lock data that is currently locked, they are blocked and have to wait. If they wait longer than the defined time out, they receive the following Microsoft Dynamics NAV error: "The XYZ table cannot be locked or changed because it is already locked by the user."

If it is necessary, change the default time-out by selecting **File > Database> Alter**. On the **Advanced** tab, select **Lock Timeout** and **Timeout duration (sec) value**.

Refer to the previous example, where two users try to modify the same record. The data that will be changed can be locked. This prevents other users from doing the same. Following is an example.

User A	User B	Comment

Customer.LOCKTABLE;
Customer.GET('10000')
;
		User A reads record with lock.

	Customer.LOCKTABLE;
	Customer.GET('10000');
		User B tries to read same record with a lock.

…	… blocked, waiting …	User B waits and is blocked, because the record is locked by user A.

Customer.Blocked :=
  Customer.Blocked::All
;
Customer.MODIFY;
	… blocked, waiting …	User A successfully modifies record.

COMMIT;		Lock is released.
	…	Data is sent to user B.

	Customer.Blocked :=
	  Customer.Blocked::"";
	Customer.MODIFY;
		User B successfully modifies record.

	COMMIT;	Lock is released.
SUCCESS	SUCCESS	

Deadlocks

There is a potential situation when blocking cannot be resolved by SQL server. The situation arises when two or more users manage to lock data. Then it is blocked when they try to lock data that is already locked by one of the other users. SQL server resolves the issue by ending the transaction that has done the least amount of work.

For example, consider a case in which two users are working at the same time and try to retrieve one another's blocked records, as shown in the following pseudo code.

User A	User B	Comment

TableX.LOCKTABLE;
TableY.LOCKTABLE;
	TableX.LOCKTABLE;
	TableY.LOCKTABLE;
		Indicates that the next read will use UPDLOCK.

TableX.FINDFIRST;	TableY.FINDFIRST;	A blocks Record1 from TableX. B blocks Record 1 from tableY.
…	…	

User A	User B	Comment
TableY.FINDFIRST;	TableX.FINDFIRST;	A wants B's record, whereas B wants A's record. A conflict occurs.
"Your activity was deadlocked with another user"		SQL Server detects deadlock and arbitrarily selects one over the other. Therefore, one will receive an error.
ERROR	SUCCESS	

SQL Server supports record level locking. So, there may be a situation where these two activities bypass one another without any problem, such as with this pseudo code. Be aware that User A is retrieving the last record compared to the situation that was discussed earlier.

User A	User B	Comment

TableX.LOCKTABLE;
TableY.LOCKTABLE;
	TableX.LOCKTABLE;
	TableY.LOCKTABLE;
		Indicates that the next read will use UPDLOCK.

TableX.FINDFIRST;	TableY.FINDFIRST;	A blocks Record1 from TableX. B blocks Record 1 from tableY.
…	…	
TableY.FINDLAST;	TableX.FINDLAST;	No conflict, as no records are in contention.
SUCCESS	SUCCESS	

Be aware that there would be a deadlock if one of the tables was empty, or contained one record only. To add to this complexity, there may be a situation where two processes read the same table from opposite directions and meet in the middle, such as with the following pseudo code.

User A	User B	Comment

TableX.LOCKTABLE;
TableY.LOCKTABLE;
	TableX.LOCKTABLE;
	TableY.LOCKTABLE;
		Indicates that the next read will use UPDLOCK.

TableX.FIND('-');	TableY.FIND('+');	A reads from top of TableX. B reads from bottom of TableX.

User A	User B	Comment
REPEAT	REPEAT	
…	…	
UNTIL NEXT = 0;	UNTIL NEXT(-1) = 0;	…after some time… A wants B's record, whereas B wants A's record. A conflict occurs.
	"Your activity was deadlocked with another user"	SQL Server detects deadlock and selects one of the users for failure.
SUCCESS	ERROR	

There are also situations where a block on index update may produce the conflict, and situations where updating SIFT tables can cause a deadlock. These situations can be complex and difficult to avoid. However, the transaction that is selected to fail is rolled back to the beginning, so there should be no major issue. However, if the process is written with several partial commits, then there might be dirty data in the database as a side-product of those deadlocks. That can become a major issue for the customer.

Avoid Deadlocks

Many deadlocks could lead to major customer dissatisfaction, but deadlocks cannot be avoided completely. To reduce the number of deadlocks, do the following:

- Process tables in the same sequence.
- Process records in the same order.
- Keep the transaction length to a minimum.

If this is not possible because of the complexity of the processes, revert to serializing the code by making sure that conflicting processes cannot execute in parallel. The following code demonstrates how you can do this.

User A	User B	Comment

TableX.LOCKTABLE;
TableY.LOCKTABLE;
TableA.LOCKTABLE;
	TableX.LOCKTABLE;
	TableY.LOCKTABLE;
	TableA.LOCKTABLE;
		Indicates that the next read will use UPDLOCK.

TableA.FINDFIRST;	…	User A locks Record1 from TableX.

User A	User B	Comment
	TableA.FINDFIRST;	User B tries to lock Record1 from TableX.
…	Blocked	User B is blocked.

TableY.FINDFIRST;
TableX.FINDFIRST;
	Blocked	User A processes tables in opposite order.

COMMIT;		Block is released.
	OK on read table A	

	TableX.FINDFIRST;
	TableY.FINDFIRST;
		User B processes tables in opposite order.

	COMMIT;	
SUCCESS	SUCCESS	

By serializing the transactions, you may experience a greater probability of timeouts. Therefore, keeping the length of transactions short becomes even more important. This also demonstrates that you can combine the various principles and methods, depending on the situation and complexity; one method may work for one customer while the other method works for another customer.

We also recommend that you consider following these *golden rules*:

- Test conditions of data validity before the start of locking.
- Allow for some time gap between heavy processes so that other users can process.
- Never allow for user input during an opened transaction.

If the transaction is too complex or if there is limited time, consider discussing the possibility of over-night processing of heavy jobs with the customer. This avoids the daily concurrency complexity and avoids the high costs of rewriting the code.

Best Practice: *In order to minimize the possibility of deadlocks (or blocks) occurring, focus first on increasing the performance of the transactions. The longer it takes for a transaction to complete, the greater the possibility for a deadlock to occur. By increasing the performance, you reduce the possibility of deadlocks occurring. Increasing performance is achieved by analyzing index usage. Make sure that indexes that are not used are not maintained. Make sure that indexes that improve the performance are available for SQL Server to use and make sure that the available indexes are optimized.*

SIFT Data Storage in SQL Server

Use SIFT tables in Microsoft Dynamics NAV version 5.0 and older, to implement SIFT on SQL Server, and to store aggregate values for SumIdexFields for keys in the source tables. Starting with version 5.0 Service Pack 1, indexed views replace these SIFT tables. SIFT tables are no longer part of Microsoft Dynamics NAV. This section is preserved because developers who work with Microsoft Dynamics NAV are likely to encounter issues about SIFT tables in implementations of older versions of Microsoft Dynamics NAV. You must have a good understanding of how they work.

A SumIndexField is always associated with a key, and each key can have no more than 20 SumIndexFields associated with it. When the MaintainSIFTIndex property of a key is set to **Yes**, Microsoft Dynamics NAV regards this key as a SIFT key and creates the SIFT structures that are needed to support it.

You can associate any field of the Decimal data type with a key as a SumIndexField. Microsoft Dynamics NAV then creates and maintains a structure that stores the calculated totals that are required for the fast calculation of aggregated totals.

In the SQL Server Option for Microsoft Dynamics NAV, this maintained structure is a typical table, but is called a SIFT table. These SIFT tables exist on SQL Server, but are not visible in the table designer in C/SIDE. As soon as you create the first SIFT table for a base table, a dedicated SQL Server trigger is also created and then is maintained automatically by Microsoft Dynamics NAV. This is known as a SIFT trigger. A base table is also a standard Microsoft Dynamics NAV table, instead of an additional SQL Server table that is created to support Microsoft Dynamics NAV functionality.

One SIFT trigger is created for each base table that contains SumIndexFields. This dedicated SQL Server trigger supports all the SIFT tables that you create to support this base table. The SIFT trigger implements all modifications that are made on the base table when a SIFT table is affected. This means that the SIFT trigger automatically updates the information in all existing SIFT tables after every modification of the records in the base table.

The name of the SIFT trigger has the following format: <base Table Name>_TG. For example, the SIFT trigger for table 17, G/L Entry is named CRONUS International Ltd_$G/L Entry_TG. Regardless of the number of SIFT keys that are defined for a base table, only one SIFT trigger is created.

You create a SIFT table for every base table key that has at least one SumIndexField associated with it. Regardless of how many SumIndexFields are associated with a key, you can create only one SIFT table for that key.

The name of the SIFT table has the following format: <Company Name>$<base Table ID>$<Key Index>. For example, one of the SIFT tables that were created for table 17, G/L Entry is named **CRONUS International Ltd_$17$0**.

The column layout of the SIFT tables is based on the layout of the SIFT key together with the SumIndexFields that are associated with this SIFT key. But the first column in every SIFT table is always named *bucket*, and it contains the value of the bucket or the SIFT level for the precalculated sums that are stored in the table. To view the structure, examine the SIFTLevels property for a key in Microsoft Dynamics NAV.

After the bucket column is a set of columns with names that begin with the letter *f*. These are also known as f- or key-columns. Each of these columns represents one field of the SIFT key.

The name of these columns has the format, f<Field No.>, where Field No. is the integer value of the Field No. property of the represented SIFT key field. For example, column f3 in CRONUS International Ltd_$17$0 represents the **G/L Account No.** field (it is field number three in the base table G/L Entry).

Finally, there is a group of columns with names that begin with the letter *s* followed by numbers. These are known as s-columns. These columns represent every SumIndexField that is associated with the SIFT key.

The name of these columns has the format, s<Field No.>. Field No. is the integer value of the Field No. property of the represented SumIndexField. The precalculated totals of values for the corresponding SumIndexFields are stored in these fields of the SIFT table.

SIFT tables are one of the biggest Microsoft Dynamics NAV performance problems on SQL Server. One record update in the base table produces a potentially large stream of Input/Output (I/O) requests with updates to the records in the SIFT tables. This could possibly block other users during that time.

SQL Server Profiler

Microsoft SQL Profiler is a graphical user interface to SQL Trace for monitoring an instance of the Database Engine. You can capture and save data about each event to a file or table to analyze later. For example, you can monitor a production environment to see which queries are affecting performance by executing too slowly.

SQL Server Profiler

SQL Server Profiler can be used to monitor events that occur on SQL Server. It can be used to do the following tasks:

- Create a trace that is based on a reusable template.
- Watch the trace results as the trace runs.
- Store the trace results in a table.
- Start, stop, pause, and modify the trace results as necessary.
- Replay the trace results.

SQL Profiler then can analyze or use the trace file to troubleshoot logic or performance problems. You can use this utility to monitor several areas of server activity, such as the following:

- Analyzing and debugging SQL statements and stored procedures
- Monitoring slow performance
- Stress analysis
- General debugging and troubleshooting
- Fine-tuning indexes
- Auditing and reviewing security activity

To summarize, you create a template or use an existing template that defines the data that you want to collect. Then you collect the data by running a trace on the events that you defined in your template. During the run, Profiler displays the event classes and data columns that describe the event data that is being collected.

SQL Server Profiler Terminology

Template

A template defines the default configuration for a trace. Templates can be saved, imported, and exported between SQL Server instances. Templates from one SQL Server version cannot be imported to a different SQL Server version. SQL Server includes the following ten predefined templates:

- **Event** – An event is an action that is generated by the SQL Server engine, such as a logon connection or the execution of a Transact-SQL statement. Events are grouped by event categories. All the data that is generated by an event is displayed in the trace. This contains columns of data that describe the event in detail.
- **Trace** – The trace does the actual data capture, based on the events that you defined in the template.
- **Event Class** – An event class can be defined as a type of event that can be traced. Examples of event classes are SP:Starting and RPC:Completed.
- **Event Category** – Groups of events are called an *event category*. Examples of event categories are Stored Procedure and Locks. There can be multiple event categories that can be selected for a single trace.
- **Data Column** – Data column is an attribute of an event class that is captured in the trace. A data column contains values of an event class.
- **Filter –** Filters are used to create selectivity in data that are collected in trace. By default, SQL Profiler monitors all events on SQL Server. You can apply a filter to only monitor events in the Microsoft Dynamics NAV database.

Use SQL Server Profiler

SQL Profiler is a component of client tools that can be installed independently from the SQL Server Database Engine. It is mandatory for the user to have system admin rights to start the profiler. You can start SQL Profiler by using the following methods:

- **Start SQL Profiler** – SQL Profiler is available from the Microsoft Windows Start menu or SQL Server Enterprise Manager. Use either of the following methods to start Profiler:
- Click Start, locate Microsoft SQL Server among your available programs, and then click Profiler on the Performance Tools group.

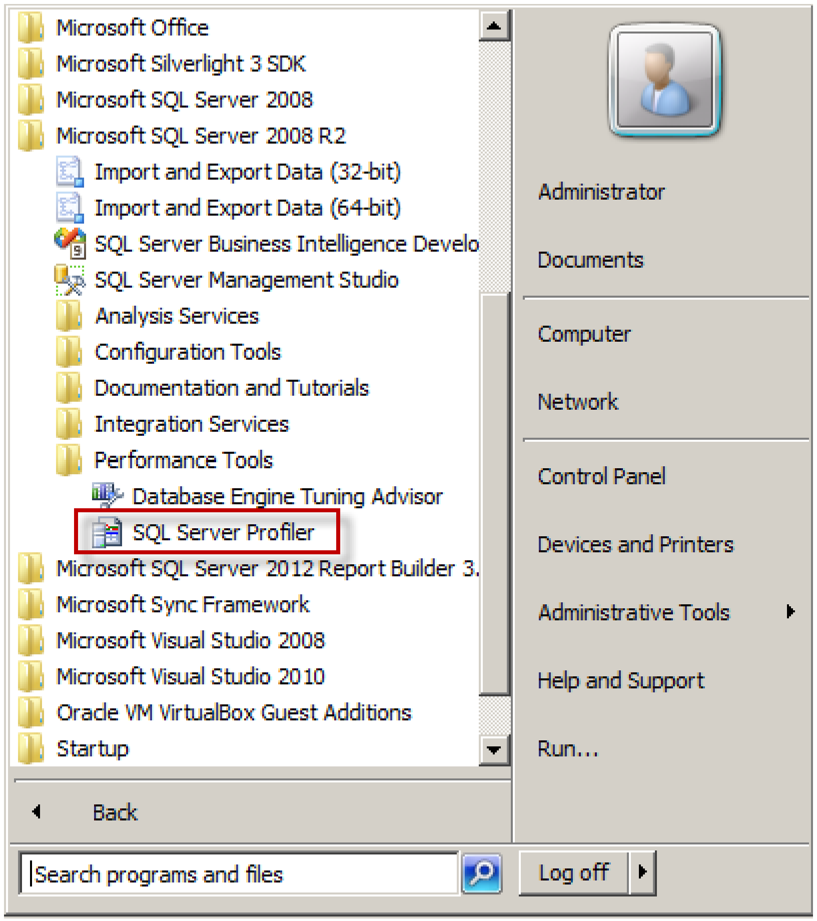

Microsoft Office
Microsoft Silverlight 3 SDK
Microsoft SQL Server 2008
Microsoft SQL Server 2008 R2
Import and Export Data (32-bit)
Import and Export Data (64-bit)
SQL Server Business Intelligence Develo
SQL Server Management Studio
Analysis Services
Configuration Tools
Documentation and Tutorials
Integration Services
Performance Tools
Database Engine Tuning Advisor
SQL Server Profiler
Microsoft SQL Server 2012 Report Builder 3.
Microsoft Sync Framework
Microsoft Visual Studio 2008
Microsoft Visual Studio 2010
Oracle VM VirtualBox Guest Additions
Startup
Back
Search programs and files
Administrator
Documents
Computer
Network
Control Panel
Devices and Printers
Administrative Tools
Help and Support
Run...
Log off

FIGURE 13.4: START SQL SERVER PROFILER FROM START MENU

- In Enterprise Manager, select **SQL Profiler** on the **Tools** menu.

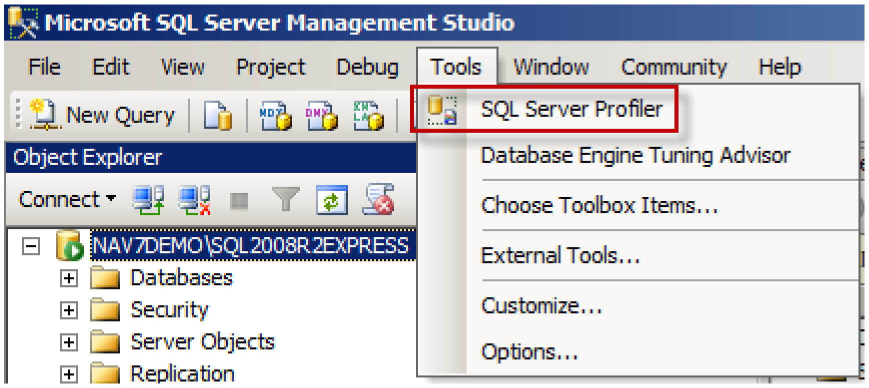
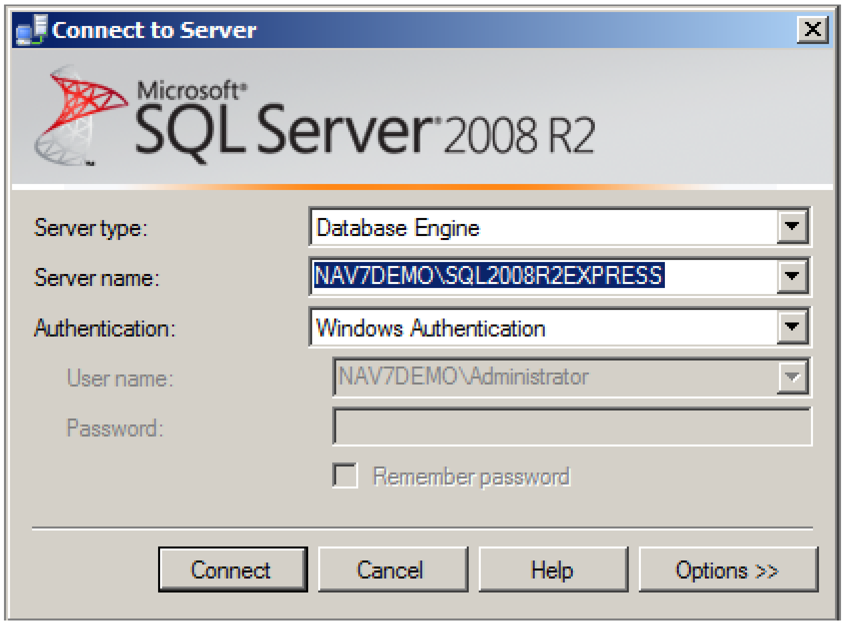


FIGURE 13.5: START SQL SERVER PROFILER FROM SQL SERVER MANAGEMENT STUDIO

- **Collect Data –** Select **Menu** > **File** > **New Trace** to create a new trace. A window opens to connect it to a database. The database can be a local database or a database that is available on the network, such as a production server. In the following example a connection is made to a SQL Instance that is named **SQL2008R2EXPRESS** on the server **NAV7DEMO**.

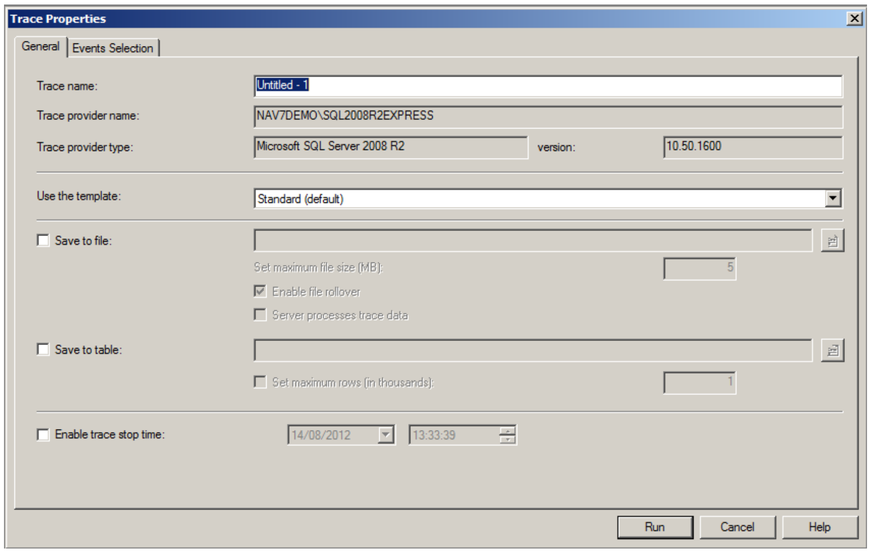

Connect to Server

Server type: Database Engine
Server name: NAV7DEMO\SQL2008R2EXPRESS
Authentication: Windows Authentication
User name: NAV7DEMO\Administrator
Password:
☐ Remember password

FIGURE 13.6: CONNECT TO SERVER WINDOW

***Note:** NoteBe aware that you must be a member of the SYSADMIN fixed server role to be able to setup traces with Profiler.*

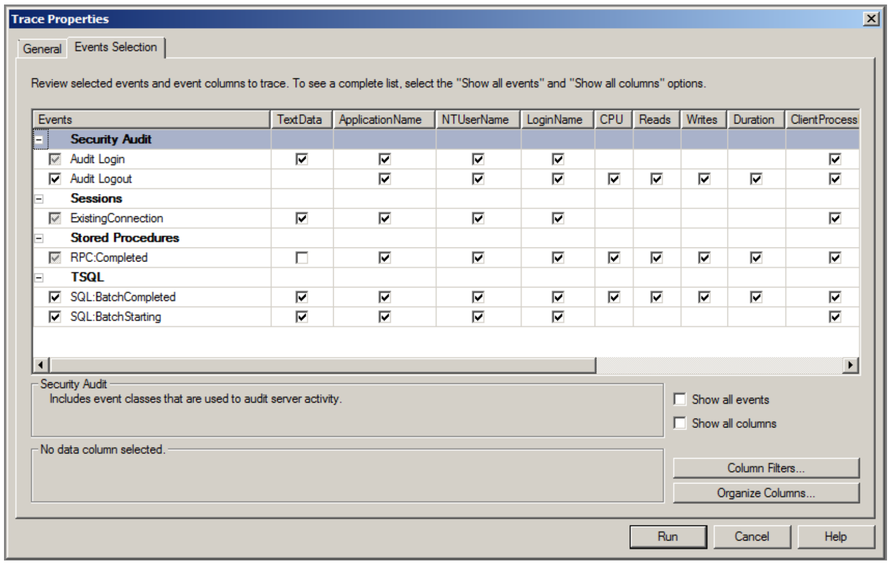
The **Trace Properties** window opens after the connection to SQL Server is made. Here you can enter the trace name. Notice that trace provider name, type, and versions are prepopulated and you cannot alter them. These are set based on the instance of SQL Server with which you are connected.


Trace Properties

Trace name: Untitled - 1
Trace provider name: NAV7DEMO\SQL2008R2EXPRESS
Trace provider type: Microsoft SQL Server 2008 R2 version: 10.50.1600
Use the template: Standard (default)
☐ Save to file:
Set maximum file size (MB): 5
☑ Enable file rollover
☐ Server processes trace data
☐ Save to table:
☐ Set maximum rows (in thousands): 1
☐ Enable trace stop time: 14/08/2012 13:33:39

FIGURE 13.7: TRACE PROPERTIES GENERAL WINDOW

The **Trace Properties** window contains a drop-down for template selection. This is named **Use the template**. The selected default template is standard. To view the events that are included in the **Standard** template, select the **Events Selection** tab.


Trace Properties

General | Events Selection

Review selected events and event columns to trace. To see a complete list, select the "Show all events" and "Show all columns" options.

Events	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration	ClientProcess
Security Audit									
✓ Audit Login	✓	✓	✓	✓					✓
✓ Audit Logout		✓	✓	✓	✓	✓	✓	✓	✓
Sessions									
✓ ExistingConnection	✓	✓	✓	✓					✓
Stored Procedures									
✓ RPC:Completed	☐	✓	✓	✓	✓	✓	✓	✓	✓
TSQL									
✓ SQL:BatchCompleted	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓ SQL:BatchStarting	✓	✓	✓	✓					✓

Security Audit
Includes event classes that are used to audit server activity.

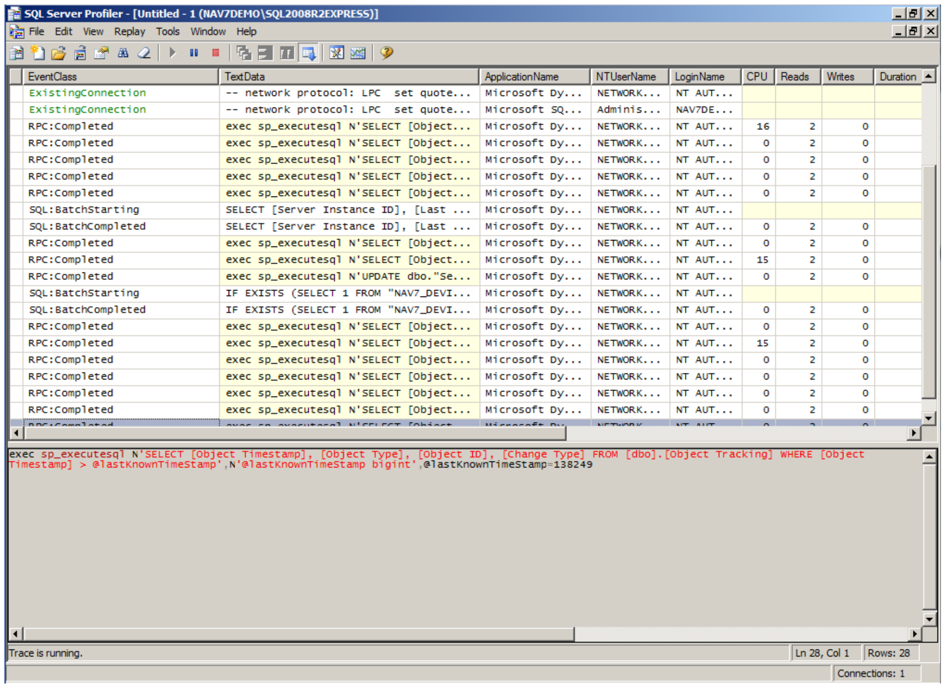
☐ Show all events
☐ Show all columns

No data column selected.

FIGURE 13.8: TRACE PROPERTIES EVENTS SELECTION

The **Event selection** window contains check boxes that can be configured, depending on the requirements of the data that you want to collect. There is also a **Column Filters** button that you can click to further filter trace data.

The **Trace Properties** window contains a **Run** button to start the trace. The trace can be paused or stopped as required by using the Play menu.


SQL Server Profiler - [Untitled - 1 (NAV7DEMO\SQL2008R2EXPRESS)]
File Edit View Replay Tools Window Help

EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration
ExistingConnection	-- network protocol: LPC set quote...	Microsoft Dy...	NETWORK...	NT AUT...				
ExistingConnection	-- network protocol: LPC set quote...	Microsoft SQ...	Adminis...	NAV7DE...				
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	16	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
SQL:BatchStarting	SELECT [Server Instance ID], [Last ...	Microsoft Dy...	NETWORK...	NT AUT...				
SQL:BatchCompleted	SELECT [Server Instance ID], [Last ...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	15	2	0	
RPC:Completed	exec sp_executesql N'UPDATE dbo."Se...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
SQL:BatchStarting	IF EXISTS (SELECT 1 FROM "NAV7_DEVI...	Microsoft Dy...	NETWORK...	NT AUT...				
SQL:BatchCompleted	IF EXISTS (SELECT 1 FROM "NAV7_DEVI...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	15	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	
RPC:Completed	exec sp_executesql N'SELECT [Object...	Microsoft Dy...	NETWORK...	NT AUT...	0	2	0	

```
exec sp_executesql N'SELECT [Object Timestamp], [Object Type], [Object ID], [Change Type] FROM [dbo].[Object Tracking] WHERE [Object
Timestamp] > @lastKnownTimeStamp',N'@lastKnownTimeStamp bigint',@lastKnownTimeStamp=138249
```

Trace is running. Ln 28, Col 1 Rows: 28

Connections: 1

FIGURE 13.9: A RUNNING TRACE

Additional Reading: *More information about SQL Profiler is available on the MSDN website that is located at: http://go.microsoft.com/fwlink/?LinkId=269809.*

Lab 13.1: Analyze Index Usage

Scenario

Mort is asked to analyze the indexes of the Cronus database. To optimize the performance of certain processes, he must make sure that indexes that are not used by SQL Server are disabled to minimize overhead. To perform this analysis, Mort executes a query. This query displays a list of all tables and indexes in an SQL database to help identify the tables in which most blocks occur. This gives a starting point for index tuning.

Objectives

Identify indexes that might cause overhead on SQL Server.

Exercise 1: Use the Index Information Query to identify and disable unused indexes.

Task 1: Execute the Index Information Query

High Level Steps

1. Open SQL Server Management Studio.
2. Execute the Query.
3. Run the Index Information Query.
4. Analyze the Query Results.

Detailed Steps

1. Open SQL Server Management Studio.
    a. To open Microsoft SQL Server Management Studio select: **Start > All Programs > Microsoft SQL Server 2012 > SQL Server Management Studio** on the main Toolbar. The **Connect to SQL Server** window opens.
    b. Select **Connect** to connect to SQL Server.
    c. To open a new Query window, select the **New Query** button (or Ctrl+N). It can be found at the top of the screen, below the menu.

2. Execute the Query.
    a. Type the following Query in the Query window.

Code Example

```
use  [Demo Database NAV (7-0)]

IF OBJECT_ID ('z_IUQ2_Temp_Index_Keys', 'U') IS NOT NULL
```

```
DROP TABLE z_IUQ2_Temp_Index_Keys;

-- Generate list of indexes with key list

create table z_IUQ2_Temp_Index_Keys

([l1] [bigint] NOT NULL,

[F_Obj_ID] [bigint] NOT NULL,

[F_Schema_Name] [nvarchar] (128) NULL,

[F_Table_Name] [nvarchar] (128) NOT NULL,

[F_Row_Count] [bigint] NULL,

[F_Reserved] [bigint] NULL,

[F_Data] [bigint] NULL,

[F_Index_Size] [bigint] NULL,

[F_UnUsed] [bigint] NULL,

[F_Index_Name] [nvarchar] (128) NULL,

[F_Index_ID] [bigint] NOT NULL,

[F_Column_Name] [nvarchar] (128) NULL,

[F_User_Updates] [bigint] NULL,

[F_User_Reads] [bigint] NULL,

[F_Locks] [bigint] NULL,

[F_Blocks] [bigint] NULL,

[F_Block_Wait_Time] [bigint] NULL,

[F_Last_Used] [datetime] NULL,

[F_Index_Type] [nvarchar] (128) NOT NULL,

[F_Index_Column_ID] [bigint] NOT NULL,

[F_Last_Seek] [datetime] NULL,

[F_Last_Scan] [datetime] NULL,
[F_Last_Lookup] [datetime] NULL,

[Index_Key_List] [nvarchar] (MAX) NULL

)

GO

CREATE NONCLUSTERED INDEX [Object_ID_Index] ON [dbo].

[z_IUQ2_Temp_Index_Keys]

(

[F_Obj_ID]

ASC

)

GO

CREATE NONCLUSTERED INDEX [Index_ID_Index] ON [dbo].

[z_IUQ2_Temp_Index_Keys]

(

[F_Index_ID]

ASC

)

GO

CREATE NONCLUSTERED INDEX [RowCount_ID_Index] ON [dbo].

[z_IUQ2_Temp_Index_Keys]

(

[F_Row_Count]

ASC

)

GO
```

```
INSERT INTO z_IUQ2_Temp_Index_Keys

SELECT

(

row_number() over(order by a3.name, a2.name))%2 as l1,

a1.object_id,

a3.name AS [schemaname],

a2.name AS [tablename],

a1.rows as row_count,

(

a1.reserved + ISNULL(a4.reserved,0))* 8 AS reserved,

a1.data * 8 AS data,

(

CASE WHEN (a1.used + ISNULL(a4.used,0)) > a1.data THEN (a1.used +
ISNULL(a4.used,0)) - a1.data ELSE 0 END) * 8 AS index_size,

(

CASE WHEN (a1.reserved + ISNULL(a4.reserved,0)) > a1.used THEN (a1.reserved +
ISNULL(a4.reserved,0)) - a1.used ELSE 0 END) * 8 AS unused,

-- Index Description

SI.name,

SI.Index_ID,

index_col(object_name(SIC.object_id),SIC.index_id,SIC.Index_Column_ID),

-- Index Stats

US.user_updates,

US.user_seeks + US.user_scans + US.user_lookups User_Reads,

-- Index blocks

IStats.row_lock_count + IStats.page_lock_count,
IStats.row_lock_wait_count + IStats.page_lock_wait_count,

IStats.row_lock_wait_in_ms + IStats.page_lock_wait_in_ms,

-- Dates

CASE

WHEN

  (ISNULL(US.last_user_seek,'00:00:00.000') >=
ISNULL(US.last_user_scan,'00:00:00.000')) and

  (ISNULL(US.last_user_seek,'00:00:00.000') >=
ISNULL(US.last_user_lookup,'00:00:00.000'))

THEN

  US.last_user_seek

WHEN

  (ISNULL(US.last_user_scan,'00:00:00.000') >=
ISNULL(US.last_user_seek,'00:00:00.000')) and

  (ISNULL(US.last_user_scan,'00:00:00.000') >=
ISNULL(US.last_user_lookup,'00:00:00.000'))

THEN

  US.last_user_scan

ELSE

  US.last_user_lookup

END AS Last_Used_For_Reads,

SI.type_desc,

SIC.index_column_id,

US.last_user_seek,

US.last_user_scan,

US.last_user_lookup,

''
```

```
FROM

(

SELECT

ps.object_id,

SUM

(

CASE

WHEN

  (ps.index_id < 2)

THEN

  row_count

ELSE

  0

END

)

AS [rows],

SUM(ps.reserved_page_count) AS reserved,

SUM

(

CASE

WHEN

  (ps.index_id < 2)

THEN

  (ps.in_row_data_page_count + ps.lob_used_page_count +
ps.row_overflow_used_page_count)

ELSE
```

```
  (ps.lob_used_page_count + ps.row_overflow_used_page_count)

END

)

AS data,

SUM (ps.used_page_count) AS used

FROM

sys.dm_db_partition_stats ps

GROUP BY ps.object_id) AS a1

LEFT OUTER JOIN

(

SELECT

it.parent_id,

SUM (ps.reserved_page_count) AS reserved,

SUM (ps.used_page_count) AS used

FROM sys.dm_db_partition_stats ps

INNER JOIN sys.internal_tables it ON (it.object_id = ps.object_id)

WHERE it.internal_type IN (202,204)

GROUP BY it.parent_id) AS a4 ON (a4.parent_id = a1.object_id)

INNER JOIN sys.all_objects a2 ON ( a1.object_id = a2.object_id )

INNER JOIN sys.schemas a3 ON (a2.schema_id = a3.schema_id)

INNER JOIN sys.indexes SI ON (SI.object_id = a1."object_id")

INNER JOIN sys.index_columns SIC ON (SIC.object_id = SI.object_id and
SIC.index_id = SI.index_id)

LEFT OUTER JOIN sys.dm_db_index_usage_stats US ON (US.object_id = SI.object_id
and US.index_id = SI.index_id and US.database_id = db_id())

LEFT OUTER JOIN sys.dm_db_index_operational_stats(NULL,NULL,NULL,NULL)
IStats ON (IStats.object_id = SI.object_id and IStats.index_id = SI.index_id and
IStats.database_id = db_id())

WHERE a2.type <> N'S' and a2.type <> N'IT'

ORDER BY row_count DESC

GO

-- Populate key string

DECLARE IndexCursor CURSOR FOR SELECT

F_Obj_ID,

F_Index_ID

FROM

z_IUQ2_Temp_Index_Keys

FOR UPDATE OF

Index_Key_List

DECLARE @objID int

DECLARE @IndID int

DECLARE @KeyString VARCHAR(MAX)

SET @KeyString = NULL

OPEN IndexCursor

SET NOCOUNT ON

FETCH NEXT FROM IndexCursor INTO @ObjID, @IndID

WHILE @@fetch_status = 0 BEGIN

  SET @KeyString = ''

  SELECT @KeyString = COALESCE(@KeyString,'') + F_Column_Name + ', '

  FROM z_IUQ2_Temp_Index_Keys

  WHERE F_Obj_ID = @ObjID and F_Index_ID = @IndID

  ORDER BY F_Index_ID, F_Index_Column_ID
  SET @KeyString = LEFT(@KeyString,LEN(@KeyString) -2)

  UPDATE z_IUQ2_Temp_Index_Keys

  SET Index_Key_List = @KeyString

  WHERE CURRENT OF IndexCursor

  FETCH NEXT FROM IndexCursor INTO @ObjID, @IndID

END;

CLOSE IndexCursor

DEALLOCATE IndexCursor

GO

-- clean up table to one line per index

DELETE FROM  z_IUQ2_Temp_Index_Keys

WHERE [F_Index_Column_ID] > 1

GO

-- Select results

SELECT

[F_Table_Name] TableName,

[F_Row_Count] No_Of_Records,

[F_Data] Data_Size,

[F_Index_Size] Index_Size,

[F_Index_Name] Index_Name,

[F_User_Updates] Index_Updates,

[F_User_Reads] Index_Reads,

CASE WHEN

F_User_Reads = 0 THEN F_User_Updates

ELSE
```

```
F_User_Updates / F_User_Reads

END AS Updates_Per_Read,

[F_Locks] Locks,

[F_Blocks] Blocks,

[F_Block_Wait_Time] Block_Wait_Time,

[F_Last_Used] Index_Last_Used,

[F_Index_Type] Index_Type,

[Index_Key_List] Index_Fields

FROM z_IUQ2_Temp_Index_Keys

--order by F_Row_Count desc, F_Table_Name, [F_Index_ID]

--order by F_User_Updates desc

--order by Blocks desc

--order by Block_Wait_Time desc

order by Updates_Per_Read desc

--ORDER BY F_Table_Name
```

   b. As an alternative you can copy/paste the query into the Query Designer from the file IndexUsageQuery.sql.

3. Run the Index Information Query.
   a. Select **Query**, and then **Execute** to run the query.

4. Analyze the Query Results.

   In this exercise we presume the results are valid and representative for the workload in Microsoft Dynamics NAV.

   a. In the Query Results the following might be an example of an index that is unused, and that is causing overhead during write statements:
      i. Table Name: **CRONUS International Ltd_$Sales Header**
      ii. Index Name: $4
      iii. Index Updates: 1246

Note: *This number might be different, because it depends on how the database was used. It will be different almost every time you run this query. The number does not need to be the same when you test it.*

iv. Index Reads: 0

v. Index Fields: Document Type, Combine Shipments, Bill-to Customer No_, Currency Code, EU 3-Party Trade, No

According to the Index Information Query this index has never been used by SQL Server in a READ operation. However, it has already been updated multiple times. To disable the overhead of updating this index when there is a write operation in the Sales Header table, we will no longer maintain this index on SQL Server.

Task 2: Disable an unused Index

High Level Steps

1. Determine the key that correspond to index $4 in the Sales Header table.
2. Disable the index on SQL Server.

Detailed Steps

1. Determine the key that correspond to index $4 in the Sales Header table.
   a. Indexes in SQL Server correspond to Keys in Microsoft Dynamics NAV. Index **$4** in SQL Server corresponds to the fifth key in the Dynamics NAV Table designer for table **Sales Header**.
   b. In the Microsoft Dynamics NAV Development Environment, open the Object Designer (if not already open).
   c. Select Tables (on the left) and scroll to table 36 "Sales Header".
   d. Click on the Design button.
   e. Select **View, Keys**.
   f. Select the fifth row and select **View, Properties**.
   g. In SQL Server Management Studio, in the Object Explorer (on the left), expand the Databases folder.
   h. In the Databases folder, expand "Demo Database NAV (7-0)" and then expand Tables.
   i. In the Tables folder, scroll down to the line: "dbo.CRONUS International Ltd_$Sales Header" and expand it.
   j. Expand the Indexes folder.
   k. Select the line that contains "$4 (Unique, Non-Clustered)" then Right-Click and select **Properties**.

l. The Index Properties window opens for index $4.

m. Verify that you can open the index properties and compare the fields in the index with the fields in the key.

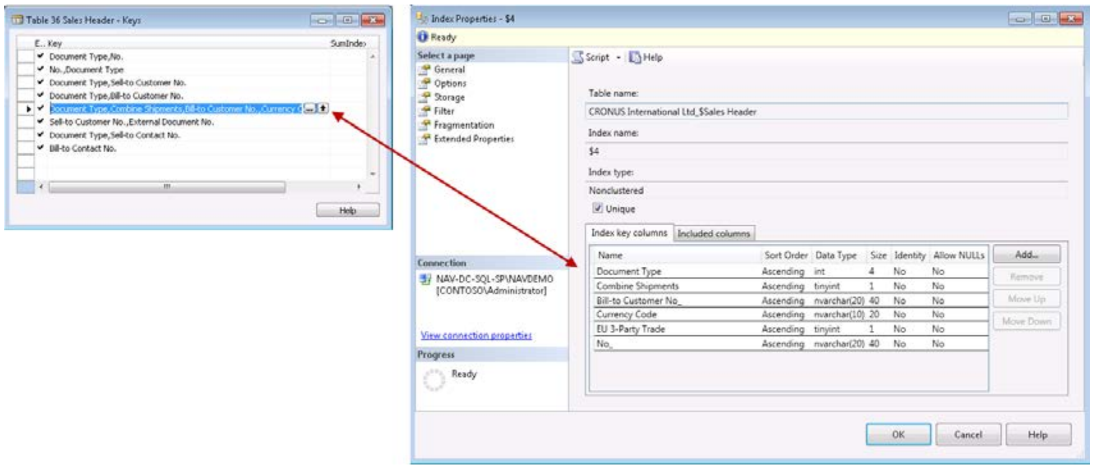


Table 36 Sales Header - Keys

E..	Key	SumIndex
✓	Document Type,No.	
✓	No.,Document Type	
✓	Document Type,Sell-to Customer No.	
✓	Document Type,Bill-to Customer No.	
✓	Document Type,Combine Shipments,Bill-to Customer No.,Currency C	
✓	Sell-to Customer No.,External Document No.	
✓	Document Type,Sell-to Contact No.	
✓	Bill-to Contact No.	

Index Properties - $4

Table name: CRONUS International Ltd_$Sales Header

Index name: $4

Index type: Nonclustered

Unique: ✓

Connection: NAV-DC-SQL-SP\NAVDEMO [CONTOSO\Administrator]

Progress: Ready

Name	Sort Order	Data Type	Size	Identity	Allow NULLs
Document Type	Ascending	int	4	No	No
Combine Shipments	Ascending	tinyint	1	No	No
Bill-to Customer No_	Ascending	nvarchar(20)	40	No	No
Currency Code	Ascending	nvarchar(10)	20	No	No
EU 3-Party Trade	Ascending	tinyint	1	No	No
No_	Ascending	nvarchar(20)	40	No	No

FIGURE 13.10: COMPARE KEY AND INDEX

2. Disable the index on SQL Server.

   a. So that you do not maintain the index in SQL Server, the property MaintainSQLIndex for the key has to be set to **No** as follows**:**

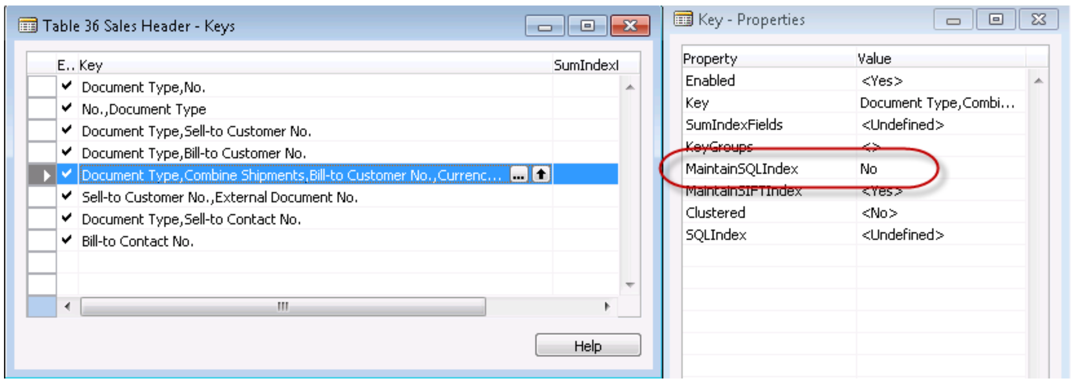


Table 36 Sales Header - Keys

E..	Key	SumIndexI
✓	Document Type,No.	
✓	No.,Document Type	
✓	Document Type,Sell-to Customer No.	
✓	Document Type,Bill-to Customer No.	
✓	Document Type,Combine Shipments,Bill-to Customer No.,Currenc...	
✓	Sell-to Customer No.,External Document No.	
✓	Document Type,Sell-to Contact No.	
✓	Bill-to Contact No.	

Key - Properties

Property	Value
Enabled	<Yes>
Key	Document Type,Combi...
SumIndexFields	<Undefined>
KeyGroups	<>
MaintainSQLIndex	No
MaintainSIFTIndex	<Yes>
Clustered	<No>
SQLIndex	<Undefined>

FIGURE 13.11: MAINTAINSQLINDEX PROPERTY

   b. Close, and then save the table.

Lab 13.2: Optimize C/AL Code

Scenario

Mort is asked to improve the performance of a code unit.

Exercise 1: Analyze and improve the C/AL code and corresponding SQL statements

Task 1: Analyze the generated SQL Statements

High Level Steps

1. Import Codeunit 123456780.
2. Design CodeUnit 123456780.
3. Analyze the C/AL code of the Batch Job and make improvements where applicable.
4. Start SQL Profiler.
5. Start a new trace.
6. Execute the Batch Job.
7. Stop, and then save the trace.
8. Analyze the Trace.

Detailed Steps

1. Import Codeunit 123456780.
   a. In the Microsoft Dynamics NAV Development Environment, open the Object Designer by selecting **Tools,** and then click **Object Designer**.
   b. Select **File**, and then click **Import**.
   c. Select the file CodeUnit 123456780.fob.
   d. Click **Open**.

2. Design CodeUnit 123456780.
   a. In the Object Designer, click **CodeUnit**.
   b. Select **CodeUnit 123456780** in the list.
   c. Click **Design** to open the C/AL Editor.

3. Analyze the C/AL code of the Batch Job and make improvements where applicable.
   a. The following code contains the batch job that updates the **Name 2** field of the **Customer** table.

Code Example

```
Customer.SETRANGE(City,'London');

Customer.FINDFIRST;

REPEAT

  Customer."Name 2" := 'Updated' + FORMAT(CURRENTDATETIME);

  Customer.MODIFY;

UNTIL Customer.NEXT = 0;

MESSAGE('Ready!');
```

4. Start SQL Profiler.
   a. Start the SQL Profiler by clicking **Start > All Programs > Microsoft SQL Server 2012 > Performance Tools > SQL Server Profiler**.

5. Start a new trace.
   a. In the **SQL Server Profiler** window, start a new trace. In the **Tools** menu, click **File**, and then **New Trace**.
   b. In the **Connect to Server** window, select **Connect**. The **Trace Properties** window opens.
   c. In the **Use the template** field, select Tuning.
   d. Click the **Events Selection** tab.
   e. Select **Column Filters** . The **Edit Filter** window opens.
   f. Select the field **DatabaseName**.
   g. In the filter window, enter the Dynamics NAV database name in the **Like** option.
   h. Select **OK**.
   i. Click **Run** to start the trace.

6. Execute the Batch Job.
   a. In the Object Designer, select the CodeUnit 123456780, and then click **Run**.

7. Stop, and then save the trace.
   a. In the SQL Profiler window, click **File**, and then **Stop Trace**.
   b. To save the trace file to the database for further analysis click: **File > Save as >Trace Table**.

c. In the **Connect to Server** window, click **Connect** to connect to SQL Server.
d. In the **Destination Table** window, in the **Database** field select the Dynamics NAV database: "Demo Database NAV (7-0)".
e. In the **Destination Table** window, in the **Table** field enter "SQLProfilerTraceResultBefore".
f. Click Ok.

8. Analyze the Trace.
   a. To open Microsoft SQL Server Management Studio, click **Start > All Programs > Microsoft SQL Server 2012 > SQL Server Management Studio**. The **Connect to SQL Server** window opens.
   b. Click **Connect** to connect to SQL Server.
   c. To open a new Query window click **New Query**.
   d. Enter the following Query to display the trace results.

Code Example

```
use [Demo Database NAV (7-0)]

SELECT * FROM SQLProfilerTraceResultBefore

WHERE TextData LIKE '%Customer%'
```

   e. Click **Query**, and then **Execute** to run the query.
   f. In the query result, you see multiple lines lines. There are several SELECT and multiple UPDATE statements.
      i. Please ignore statements that don't begin with SELECT or UPDATE.

   g. In the beginning there's a SELECT statement that retrieves one customer record with a WITH(READUNCOMMITTED) at the end.
   h. The next SELECT statement on the Custmer table, retrieves the customer record(s) to be modified with a TOP X and uses a WITH(UPLOCK) statement.
   i. Optimize the C/AL Code so that only one SELECT statement is generated and executed.

Task 2: Optimize the C/AL Code

High Level Steps

1. In the C/AL code of the imported CodeUnit (123456780) add a **SETCURRENTKEY** function.
2. Change **FIND('-')** to **FINDSET.**
3. Save, and then compile the changes to the CodeUnit.

Detailed Steps

1. In the C/AL code of the imported CodeUnit (123456780) add a **SETCURRENTKEY** function.
   a. Add a **SETCURRENTKEY(City)** statement before executing the **FIND** function.

2. Change **FIND('-')** to **FINDSET.**
   a. Because the code is looping over **Customer** records, and inside the loop it is modifying the records, you should use a FINDSET(TRUE,FALSE) statement. This applies the correct isolation level to the records. This avoids creating a second SELECT statement to lock the records.
   b. Also, use an IF statement to only run the update when there are customer records.
   c. The code now resembles this:

```
Customer.SETCURRENTKEY(City);

Customer.SETRANGE(City,'London');

IF Customer.FINDSET(TRUE,FALSE) THEN

REPEAT

  Customer."Name 2" := 'Updated' + FORMAT(CURRENTDATETIME);

  Customer.MODIFY;

UNTIL Customer.NEXT = 0;

MESSAGE('Ready!');
```

3. Save, and then compile the changes to the CodeUnit.
   a. Save the changes to the CodeUnit by selecting **File,** and then **Save**.
   b. Close the CodeUnit.

Task 3: Analyze the generated SQL Statements after Optimization

High Level Steps

1. Start a new SQL Profiler Trace.
2. Execute the optimized CodeUnit.
3. Export the Trace results.
4. Analyze the Trace.

Detailed Steps

1. Start a new SQL Profiler Trace.
   a. Start a new trace in SQL Profiler by using the same Trace Template and DatabaseName filter.

2. Execute the optimized CodeUnit.
   a. In the Object Designer, select the CodeUnit 123456780 and click **Run**.

3. Export the Trace results.
   a. Stop the Trace in SQL Profiler.
   b. Export the Trace Results to a new table named **SQLProfilerTraceResultAfter**.

4. Analyze the Trace.
   a. Open a new Query window by click **New Query**).
   b. Enter the following query to display the trace results.

Code Example

```
use [Demo Database NAV (7-0)]

SELECT * FROM SQLProfilerTraceResultAfter

WHERE TextData LIKE '%Customer%'
```

   c. Click **Query** and then **Execute** to run the query.
   d. In the query result, you see multiple lines. There are one SELECT and four UPDATE statements.
      i. Please ignore statements that don't begin with SELECT or UPDATE.
   e. The SELECT statement retrieves the customer records with a WITH(UPDLOCK) at the end.
   f. Notice that by optimizing the code the same result is obtained with less SQL statements and more optimal locking of tables.

Module Review

Module Review and Takeaways

This module covered the major points of insuring optimal performance in Microsoft Dynamics NAV applications, especially when you use SQL Server.

The specific ways in which SQL Server is implemented were discussed in detail to reveal why some operations are expensive as measured by performance, and other operations are not as expensive and just as effective.

The labs demonstrated how to use the SQL Profiler to analyze what happens on SQL Server. This tool should be used sparingly in a production environment, since the tool itself consumes a large number of system resources.

Test Your Knowledge

Test your knowledge with the following questions.

1. What are two important proprietary Dynamics NAV features that are simulated on SQL Server? Explain how these features are simulated.

2. What is the purpose of collation?

3. How can a SQL Server index be disabled from the table designer in C/SIDE without disabling the key?

4. What can be done to help avoid deadlocks?

5. Explain the difference between the clustered index and the primary key.

6. How is SIFT stored on SQL Server?

7. What tools in SQL Server can be used to troubleshoot performance issues?

Test Your Knowledge Solutions

Module Review and Takeaways

1. What are two important proprietary Dynamics NAV features that are simulated on SQL Server? Explain how these features are simulated.

   MODEL ANSWER:

   SIFT, which is simulated on SQL Server by indexed views, and before version 5.0 SP1 by additional SIFT tables.

   Data Versioning, which is simulated on SQL Server by including a datetime value for each record in the database.

2. What is the purpose of collation?

   MODEL ANSWER:

   The collation of a database determines which character set is used to store the values in the database. It determines the way that data is sorted, and can affect the way that data is retrieved from the database.

3. How can a SQL Server index be disabled from the table designer in C/SIDE without disabling the key?

   MODEL ANSWER:

   By turning off the MaintainSQLIndex property of the key.

4. What can be done to help avoid deadlocks?

   MODEL ANSWER:

   To help avoid deadlocks, you can do the following:

   • Lock tables in the same order for different types of transactions.

   • Process records in the same order for different types of transactions.

   • Keep transaction length to a minimum.

   • Serialize the transaction, by locking a general table at the start of every transaction.

5. Explain the difference between the clustered index and the primary key.

   MODEL ANSWER:

   The clustered index is the index that is used by SQL Server to physically store the data. If the clustered index is set to any particular field, then SQL Server physically stores the records in the table in the order of that field.

   The primary key is the key that defines the uniqueness of a record. Primary key field values uniquely identify a record in the table.

   You can set an index other than the primary key index as a table's clustered index.

6. How is SIFT stored on SQL Server?

   MODEL ANSWER:

   Before version 5.0 SP1, SIFT was stored on SQL Server in separate SIFT tables, and SIFT totals were updated by table triggers on SQL Server. From version 5.0 SP1 forward, SIFT is stored on SQL Server by indexed views.

7. What tools in SQL Server can be used to troubleshoot performance issues?

   MODEL ANSWER:

   The SQL Profiler.