Microsoft Dynamics NAV 2013 SQL Readiness Training

Module 3: Data Access

Microsoft

Version 1.0

Released: October 1, 2012

Conditions and Terms of Use

Microsoft Confidential

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

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

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

Copyright and Trademarks

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

Table of Contents

Data Access

In this module, we will:

- Discuss **Data Access** in **Microsoft Dynamics NAV 2013**.
- Take a close look at how the various Data Access features of C/AL are implemented on the Microsoft Dynamics NAV Service Tier (NST) and how that translates into SQL Server.
- Include a discussion about the advantages and disadvantages of the following functions:
    - FIND('-')
    - FINDSET
    - FINDFIRST
    - FINDLAST
    - ISEMPTY
- Compare how the previous C/AL functions work in Microsoft Dynamics NAV 2013 as compared with earlier versions of the product, more specifically Microsoft Dynamics NAV 2009 releases.
- Compare using these **Record API** functions with new Query object, which has been introduced in **Microsoft Dynamics NAV 2013**.

What You Will Learn

After completing this Module, you will be able to:

- Understand NST data access design in **Microsoft Dynamics NAV 2013** and know when to employ which method.
- Understand some of the important differences when compared to older versions of Microsoft Dynamics NAV.

Lesson 1: Data Access Provided by SQL Server

This section provides background information on the means of data access that SQL Server provides. It is meant as context to understand the methods employed by the NST to implement the C/AL data access functions.

There are several means, which an application can employ, provided by SQL, to access data from the database. They are default result sets, server side cursors, and MARs. A brief description of each is provided in this lesson.

Default Result Sets

The simplest and most basic way of returning results is known as default result sets. When a request is submitted for execution, SQL Server sends result sets back to clients in the following way:

1. SQL Server receives a network packet from the client containing the Transact-SQL statement or batch of Transact-SQL statements to be executed.
2. SQL Server compiles and executes the statement or batch.
3. SQL Server begins applying the rows of the result set, or multiple result sets from a batch or stored procedure, in network packets and sending them to the client.

   SQL Server puts as many result set rows as possible in each packet until the network buffers are full.

   It will continue filling network buffers as the client pulls packets out on the other end.
4. The packets containing the result set rows are cached in the network buffers of the client.

   As the client application fetches the rows, the ODBC driver or the OLE DB provider pulls the rows from the network buffers and transfers the data to the client application.

   The client retrieves the results one row at a time in a forward direction.

This is also referred to as **Firehose Cursor** or **Default Result Set**.

Server Side Cursors

With versions of SQL Server prior to SQL Server 2005, only two mechanisms existed for ODBC applications like Microsoft Dynamics NAV to access result sets from the database engine, i.e. Default Result Sets and Server Side cursors.

With default result sets, the connection to SQL Server could only work with a single result at a time, i.e. the entire request had to be processed and no other requests could be made on the same connection until the original result set had been processed completely.

Another option is to use **Server Side Cursors**, which provides a means for SQL Server to handle multiple requests on a single connection, by time multiplexing. Using this approach, the SQL Server engine retrieves rows only as they are read by the application rather than returning the entire result set in a single operation.

Unlike default result sets, processing on SQL Server (for the relevant connection) does not continue between requests. Thus, this approach requires the SQL Server database engine to keep enough state in order to resume reading records in the right order when the application comes back for more data.

As the Server Side Cursor approach retrieves a row at a time (or perhaps a small block of rows at a time), it enables the SQL Server database engine to switch between multiple requests quickly on the same connection, thus simulating parallelism.

However, there are significant drawbacks to using Server Side Cursors:

- Every time the connection fetches a row (or a small block of rows) from SQL Server a network roundtrip is required.
- No further code is executed within SQL Server for the relevant connection between fetch operations.
- Different SQL query plans are often chosen.
- It tends to perform slower than a default result set (**Firehose** Cursor).

MARS

Multiple Active Result Sets (MARS) was introduced in SQL Server 2005 and, as the name implies, it enables the client to have multiple active default result sets open on the same connection simultaneously.

MARS enables the interleaved execution of multiple requests within a single connection. That is, it allows a batch to run, and within its execution, it allows other requests to execute.

 **Note:** MARS is defined in terms of interleaving, not in terms of parallel execution.

The MARS infrastructure allows multiple batches to execute in an interleaved fashion, though execution can only be switched at well-defined points.

An important difference between MARS and server-side cursors is that execution can continue on the SQL Server connection between fetches as long as the client application is continually reading and the SQL Server, server-side buffers (which contain result sets while

they are being transferred to the client) do not fill up. This way the performance is generally better with MARS because the client receives the data as quickly as possible.

Data Access before Microsoft Dynamics NAV 2013

Before looking at how Microsoft Dynamics NAV 2013 has improved data access, let us review how FIND/FINDSET should ideally have been used with earlier versions of Microsoft Dynamics NAV 2009.

FIND('-')

Usage/Scenarios

With Microsoft Dynamics NAV 2009, FIND('-') should be used ***when requesting a set of records that may not be completely enumerated***.

For example, before posting a general journal batch, all journal lines need to be checked for validity and to make sure that they balance. This requires code, which will loop through journal lines.

However, after finding the first line with an error, it is not necessary to loop through the rest. Therefore, if an error is found in the first record then the rest of the result set would not be read.

Another good example of where to use FIND('-') with earlier versions of Dynamics NAV 2009 is when you need coverage of unknown amounts, e.g. when fulfilling multiple outstanding orders from a recently received shipment. In this scenario, the number of items in the received shipment could fulfill zero or more outstanding sales orders. The actual number covered is unknown beforehand.

A final example of where it would be appropriate to use FIND('-') with earlier versions of Dynamics NAV 2009 is when you need to identify "top" amounts, e.g. the Top 10 items ordered by unit cost.

Implementation:

With previous versions of Microsoft Dynamics NAV 2009, when a FIND('-') command was being executed server-side cursors would be utilized, pre-fetching 50 records at a time.

The following illustrates the interaction between the Microsoft Dynamics NAV middle tier and SQL Server:

Table 1: Interaction between Microsoft Dynamics NAV and SQL Server

Microsoft Dynamics NAV Client or Service Tier Action	SQL Server Response
Issues request (for block of 50 rows) using server-side cursor	Retrieves 50 rows, sends them to NAV and then waits.
Reads/processes rows	Does nothing waiting for next request
Finishes with first 50 rows. Requests next 50 rows	Resumes processing from saved state and retrieves 50 more rows

FINDSET

Usage/Scenarios

With Microsoft Dynamics NAV 2009, the FINDSET command should be used ***when requesting a confined set of records that is expected to be fully (or almost fully) enumerated*.**

The initial record set returned for a FINDSET command is confined by the C/Side **Record Set** property, which can be found by:

1. Select the **File** menu option in the classic client
2. Selecting **Database** and selecting **Alter.**
3. Selecting the **Advanced** tab (see Record Set value under the **Caching** section).

If the initial record set does not contain all the required records, then Microsoft Dynamics NAV will automatically request the remaining records using the same approach as FIND('-').

FINDSET should not be used to retrieve a small number of records relative to the value configured in the **Record Set** property.

For example, if **Record Set** is specified as 1,000 rows, then FINDSET would not perform efficiently for retrieving just 100 rows. For optimum performance, the number of rows typically read with FINDSET should be several hundred (closer to 1,000) when **Record Set** is configured high.

A suitable scenario for using FINDSET (assuming an appropriately set **Record Set** value) would be as part of code to re-assign all of a specific Salesperson's customers to other Salespeople. In this scenario, you know in advance that all records retrieved will be read and processed.

Implementation

The reason why FINDSET is more efficient than FIND for confined sets is that it uses the default result set to retrieve the first X rows, where x is defined in the **RecordSet** property as mentioned earlier. This avoids the inefficiencies of server-side cursors.

However, it would be inappropriate to use FINDSET if the query were likely to return a result set with more records than are defined by the **Record Set** property. This is because, when you exceed the maximum record set size in this way then NAV will revert to using server-side cursors for the remaining records.

In certain circumstances, you may also see blocking because the default result set will retain SQL locking resources on the initial result set while NAV is working through the remaining records using the server-side cursor.

Another important performance consideration when using FINDSET with earlier versions of Microsoft Dynamics NAV 2009 is that, if the ForUpdate argument is set to true (i.e. FINDSET(TRUE)) then it will behave exactly like FIND by using server-side cursors for the entire result set.

The following illustrates the interaction between the middle tier and SQL Server when using FINDSET (without specifying TRUE for the ForUpdate parameter) with Microsoft Dynamics NAV 2009.

The value **X** represents whatever value is configured for the **Record Set** property in NAV:

Table 2: Interaction between Microsoft Dynamics NAV and SQL Server

Microsoft Dynamics NAV Client or Service Tier Action	SQL Server Response
Issues request (for up to X rows)	Retrieves up to X rows and finishes processing completely.
If a requests is made for row X + 1 then it uses a server-side cursor	Retrieves 50 rows and sends them to NAV and then waits.
From this point on the processing is similar to FIND('-') – see Table1.	

Data Access in NAV 2013

FIND('-')

Usage/Scenarios

FIND('-') should be used in Microsoft Dynamics NAV *2013* ***when requesting a set of data from a single table, which may not be completely enumerated/read***. This is similar to, but not exactly the same as, how FIND('-') was used in earlier versions of Microsoft Dynamics NAV.

The difference is that FIND('-') should now primarily be used for single table access in Microsoft Dynamics NAV 2013, since the new Query object is much more efficient for accessing data across multiple tables, especially in read-only scenarios.

Some of the examples described for FIND('-') in previous Dynamics releases are therefore still valid in 2013. The before posting a general journal batch checks are still a good example as well as when fulfilling multiple outstanding orders from a recently received shipment.

However, it would no longer be efficient to use FIND('-') to retrieve the top set of records based on a sorting criteria, e.g., to identify the Top 10 items ordered by unit cost.

The new Query object which has been introduced in Microsoft Dynamics NAV 2013 contains an explicit **TOP** property, which is highly efficient for this purpose and provides more flexibility, since the result set can be ordered by an aggregation too.

Implementation

The C/AL FIND function utilizes MARS in its underlying implementation to fulfill the result set requested.

The way the TOP clause is used with FIND includes a self-tuning optimization in Microsoft Dynamics NAV 2013. Initially, the SQL query generated will select the **TOP** 50 qualifying records.

However, if the queries being submitted are routinely returning less than 50 records then the TOP command will use a smaller value, (i.e., it is self-tuning based on statistics).

If the Microsoft Dynamics NAV middle tier attempts to read beyond the first 50 records returned (could be less than 50 records if the TOP command has been self-tuned as described previously), then a second SQL Server query is executed to retrieve all the remaining qualifying records in one SQL query (i.e., no TOP command is used for the second SQL query).

FINDSET

Usage/Scenarios

FINDSET should be used in Microsoft Dynamics NAV 2013 *when requesting a set of records from a single table that will be read in its entirety*.

For example, if you need to perform an action which requires reading or modifying every row in a record set.

The following code structure/paradigm is a classic example of where FINDSET should be used, since all records are always touched:

```
IF (<Rec>.FINDSET) THEN
REPEAT
      //(Some code).
      // (This code should not break the loop)
UNTIL (<Rec>.NEXT = 0)
```

This type of looping structure could be used to reassign a salesperson's customers. Since FINDSET is optimized for reading the entire set, this example works, even if there are a large number of customers

Implementation

A FINDSET function call utilizes MARS. The NST will not place any constraints on the SQL query, i.e., it will retrieve all rows, regardless of the size of the result set.

Summary

The following shows how FIND('-') and FINDSET are implemented with Microsoft Dynamics NAV 2013 as compared with Microsoft Dynamics NAV 2009 (and earlier) versions.

	Earlier version of Microsoft Dynamics NAV		Microsoft Dynamics NAV 2013	
	Use	Implementation	Use	Implementation
FIND('-')	Reading partially enumerated result sets	Server-Side Cursors	Reading partially enumerated result sets from single table	MARS + self-tuning TOP optimization
FINDSET	Reading fully enumerated confined sets	Default result set + Server-Side Cursors	Reading fully enumerated set (regardless of size)	MARS

Figure 1:

Other Data Access Functions

FINDFIRST and FINDLAST

There is no change to the recommendations for how FINDFIRST and FINDLAST should be used with Microsoft Dynamics NAV 2013 as compared to earlier versions of the product.

FINDFIRST should be used when you are interested in *finding only single row, which happens to be first*. It is conceptually the same as running a SELECT TOP 1 query.

If the intention is to enumerate through more than one row of the results set then you should use FIND('-') instead.

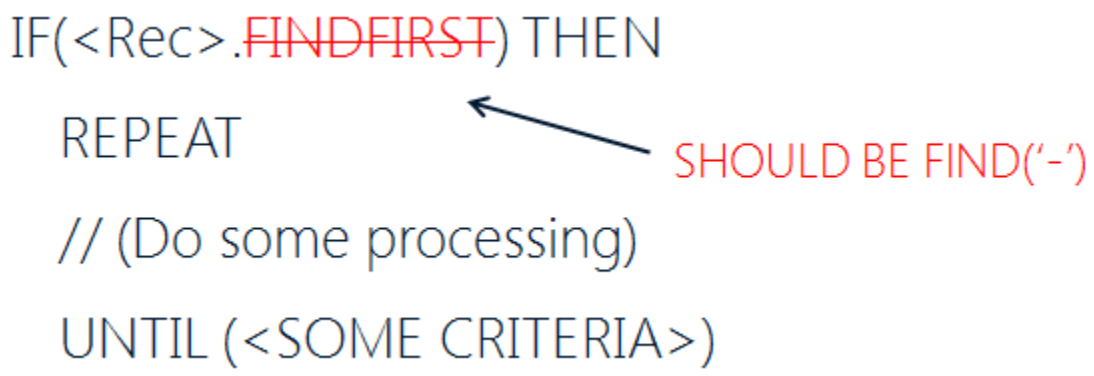

Figure 2:

Correspondingly, FINDLAST should be used when you are interested in *finding only a single row, which happens to be last*.

ISEMPTY

There is no change to the recommendations for how the C/AL ISEMPTY function should be used with Microsoft Dynamics NAV 2013 as compared to earlier versions of the product. The ISEMPTY function should be used to *identify if any qualifying row exists* when there is no need to retrieve row values. This C/AL function will be implemented as a SQL Server EXISTS query as this can identify existence of a row based on the query criteria and will use a highly efficient covering index where possible.

Optimizations

Microsoft Dynamics NAV will try to optimize if a FIND('-')/FINDSET statement is executed three times in a row without returning any records. The optimization used in this case will be different depending on the version of the product.

With earlier versions of Microsoft Dynamics NAV 2009, the subsequent executions will be converted to a SQL Server EXISTS query. If the EXISTS query returns TRUE (indicating that qualifying rows do exist) then a second SQL Server query will be executed to return the qualifying rows.

With Dynamics NAV 2013, a single optimized SQL statement will be executed, i.e., an IF EXISTS, THEN SELECT statement. This form of query will avoid a second round trip to the SQL Server.

Lesson 2: FIND Functions vs. NAV Query Objects

In Microsoft Dynamics NAV 2013, we have now introduced another means for data access, the Query object. When considering whether to use this NAV Query object to retrieve information from SQL Server or the FIND('-')/FINDSET (i.e., Record API) functions, it is important to consider the unique characteristics of each approach.

For example, if the same data is likely to be read multiple times then the Record API would be more appropriate as it can benefit from service tier data caching. As you may recall from the Query training session, Microsoft Dynamics NAV 2013 does not cache the NAV query result sets.

The Record API is also the clear winner if you need to modify data because Query object results are read-only. However, it may be efficient to use the results of a Query object to identify the records, which needs to be updated and then use the Record API calls to perform the update.

In scenarios where you only need to access a single table and want to retrieve most/all the fields, then the Record API will be a good candidate. However, much better performance can be achieved with Query objects if you need to access multiple tables, as the Query will enable you to use SQL Server's highly optimized mechanism for joining multiple tables into a single result set.

A Query object also enables the C/AL programmer to efficiently access a small subset of fields from the target tables making it easier to benefit from a covering index strategy.

Query objects also provide a means to filter and aggregate data in a way which would be much less efficient if implemented using the Record API.

Finally, Query objects can handle large quantities of data more efficiently than the Record API.

Lesson 3: Connection Pooling

Introduction

Connection Pooling is an optimization method for using SQL Server connections more efficiently. Instead of requesting a new connection to be opened for every transaction, a pool of "ready-to-go" connections are maintained to be available for use immediately. The biggest advantage of this technique is amortizing the high cost of opening a connection, since every transaction will no longer wait for the entire connection creation time.

Implementation Details

With previous versions of Microsoft Dynamics NAV (2009 and 5.0 releases), whenever a user connects to the classic client or RTC client, a connection to SQL server is established for the user session by the fat client or NAV Service Tier (NST), respectively. This connection will remain open for the duration of the client session, regardless of whether the user is active or not. The connection is closed only when the user closes the client.

For most Microsoft Dynamics NAV client sessions, the connection configuration would be very similar. Hence, during the online day, many similar connections are repeatedly opened and closed as various users log in and out of the client. Creating a new connection each time a user logs in is relatively expensive.

A connection to SQL Server consists of a series of time-consuming steps. A physical channel such as a socket or a named pipe must be established, the initial handshake with the server must occur, the connection string information must be parsed, the connection must be authenticated by the server, checks must be run for enlisting in the current transaction, and so on.

This approach does not leverage the similarity in connection configuration or the idleness of connections to optimize the process of establishing connections with SQL Server. To minimize the cost of opening connections, an optimization technique called *Connection Pooling* is used to manage NST connections to SQL Server in Microsoft Dynamics NAV 2013.

Connection pooling reduces the number of times that new connections need to be opened. The ADO.NET *Pooler* maintains ownership of the physical connection. It manages connections by keeping a set of active connections alive for each given connection configuration.

Whenever a user calls **Open** on a connection, the Pooler looks to see if there is an available connection in the pool. If a pooled connection is available, it returns it to the caller instead of opening a new connection. When the application calls **Close** on the connection, the Pooler returns it to the pooled set of active connections, instead of actually closing it.

Once the connection is returned to the pool, it is ready to be reused on the next **Open** call.

Only connections with the same configuration can be pooled. ADO.NET keeps several pools concurrently, one for each configuration.

Connections are separated into pools by connection string, as well as by Windows identity when integrated security is used.

With Microsoft Dynamics NAV 2013, the same middle tier credentials are used to access the SQL Server on behalf of all RTC client connections. Therefore, the Microsoft Dynamics NAV middle tier can maintain a pool of identical connections to SQL Server, all of which can be available for any user transaction.

When a user starts an operation, (e.g., Posting an Invoice) the NST will look for a free SQL Server connection in the Microsoft Dynamics NAV connection pool. If it finds one, it will use it, thus avoiding the overhead of creating a new SQL Server connection.

If no free SQL Server connection is available in the connection pool, the NST will establish a new SQL Server connection. However, when Posting operation is complete, the new SQL Server connection will not be closed but will be added to the connection pool. Therefore, over time the connection pool will continue to grow until it is large enough to cater for peak system load. All the connections in the pool will be maintained until the NST is stopped.

An interesting and complex example of how Connection Pooling works is with a background Posting process, which uses the job queue. When a user starts the Posting process the current user thread (thread #1) places an entry to Post into the job queue and then starts another thread (thread #2) to poll the job queue and process the Posting operation.

If there is an available SQL Server connection in the Connection Pool, thread #2 will use it rather than creating a brand new one. If the same User posts another document quickly, thread #2 may not have finished the earlier Posting process by the time a second Posting process is put in the job queue.

In this scenario, thread #1 will start another thread (thread #3) to poll/process the latest job queue entry. Therefore, a single user doing multiple background Postings could be responsible for three or more threads each with a different SQL Server connection.

If you had multiple users using the job queue simultaneously and frequently, the effect described previously could be multiplied, e.g., ten users actively submitting background posting processes could consume 30 threads with 30 SQL Server connections. Each user would require its own set of threads to ensure that items get posted in the current user context.

 **Important:** Once any SQL Server connection is established by the NST it will not be closed when the operation it executes has completed. Instead, it will remain in the NST Connection Pool, available for other subsequent operations. These SQL connections will only be released when the NST service is stopped.

Summary

When the NST is started, the number of connections in the pool is one. As more clients connect to the NST and start work that requires a connection to SQL server, more connections are opened and subsequently added to the pool. Preference goes to using an existing or idle connection over creating a new one.

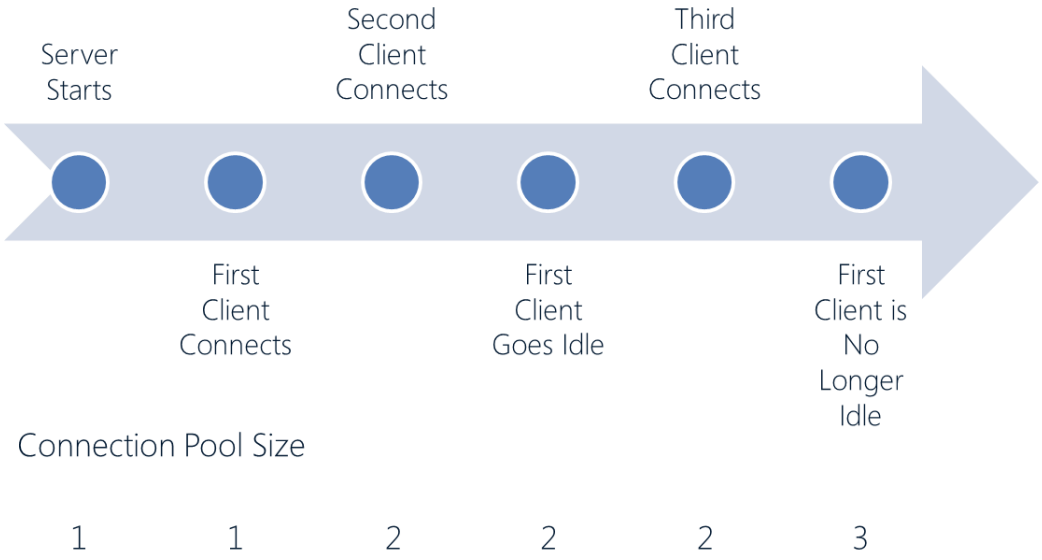

Figure 3:

More: For more information on Connection Pooling, see:

http://msdn.microsoft.com/en-us/library/8xx3tyca.aspx

Lesson 4: What's New in the Data Access Layer

The three-tier architecture was introduced with Microsoft Dynamics NAV 2009.

In Microsoft Dynamics NAV 2013, the three-tier architecture is the only runtime architecture available, as the classic runtime client has been removed. With focus on the three-tier architecture, the data access layer has been improved, optimized, and moved to managed code. In addition, the ADO.NET API is used to access SQL Server rather than ODBC.

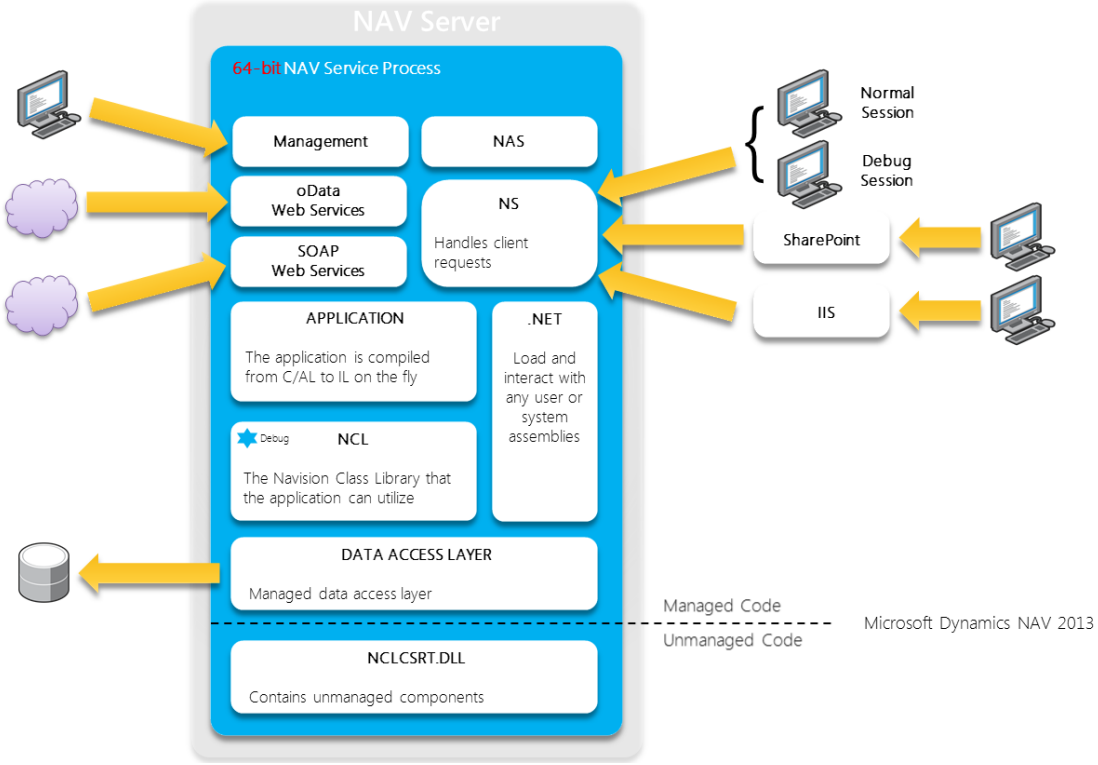

Figure 4: General Architecture

Caching

The following are the software caches, which are used by the Microsoft Dynamics NAV Service Tier (NST):

- **Primary Key Cache**—This stores a single row from a table, such as the returned result of <Rec>.GET function.
- **Result Set Cache**—As the name suggests, caches result sets, such as the returned result of <Rec>.FINDSET function.
- Calculated Field Cache
- Command Cache

While the first three of the previous caches existed in Microsoft Dynamics NAV 2009 R2, a major innovation for Microsoft Dynamics NAV 2013 is the introduction of a Global version of these three caches. Therefore, there is a Private and a Global version of each of these caches in Microsoft Dynamics NAV 2013.

Depending on whether the data has been committed to the database, results will be kept in either the Private or the Global version of the cache (see Figure 5 and Figure 6).

In the Private cache, results are only accessible by the user who created them while in the same Company. With the Global cache, the results are accessible for any user within the same Company.

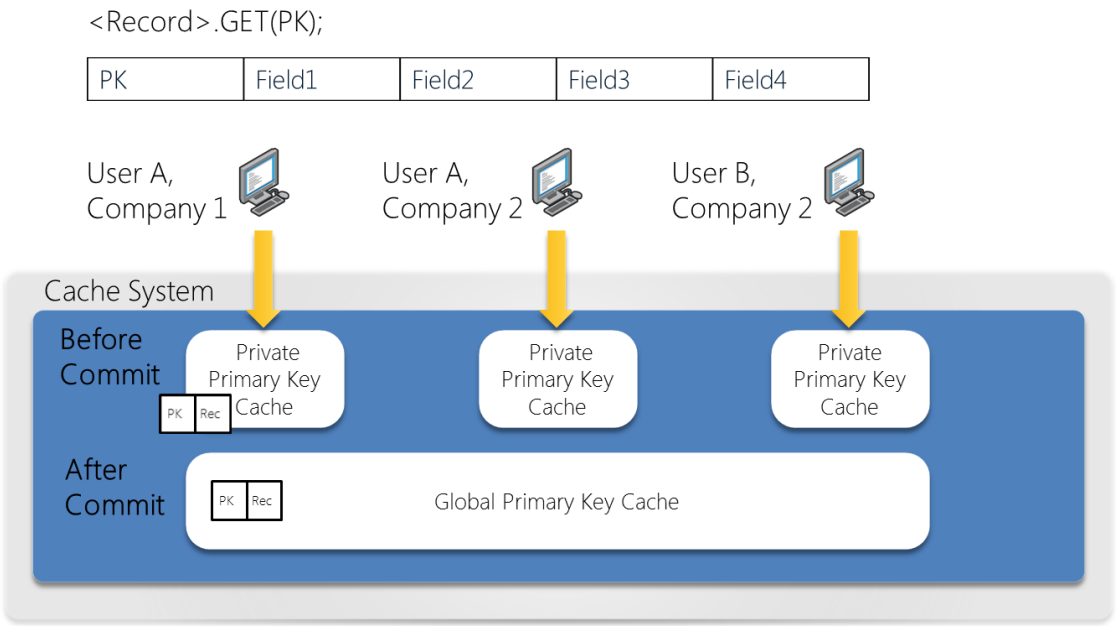

Figure 5: Primary Key Cache

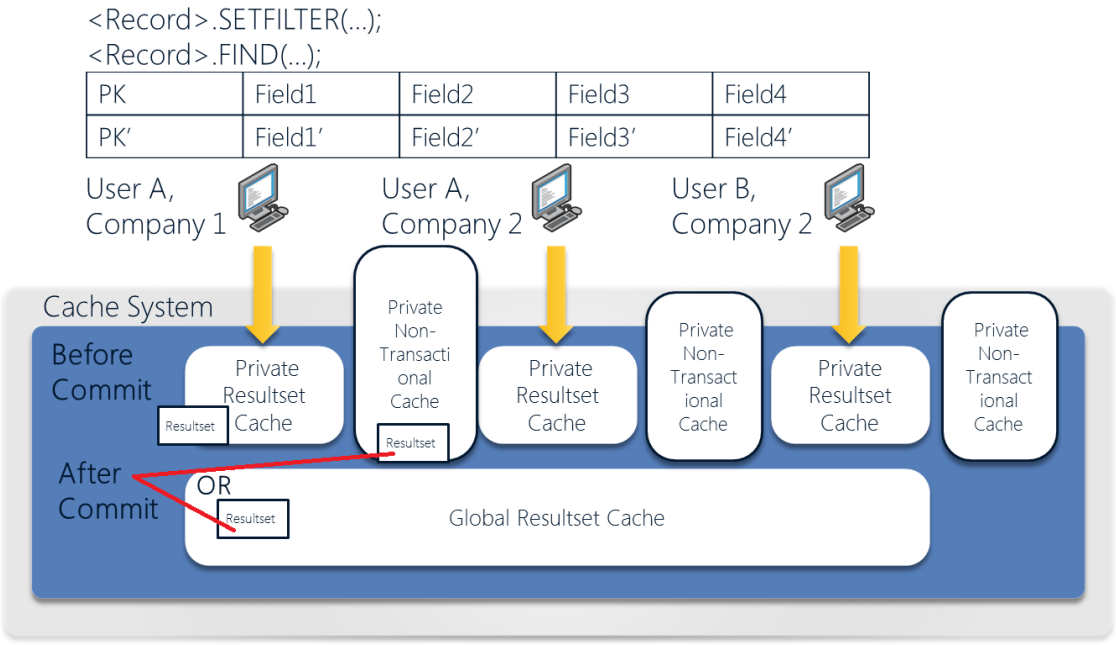

Figure 6: Result Set Cache

New Security Model

For simplicity, in Microsoft Dynamics NAV 2013, the NST uses a dedicated account (Service Account) to connect to SQL Server. This dedicated account has access to all objects. Specifically, account has the following database roles in the Microsoft Dynamics NAV database: **db_datareader, db_datawriter and db_ddladmin**. Therefore, Microsoft Dynamics NAV users are no longer created as SQL logins with associated database users in SQL Server.

The Enhanced and Standard security models, which exist in previous versions of Microsoft Dynamics NAV have been removed in Microsoft Dynamics NAV 2013. With the new security model, the Microsoft Dynamics NAV Role permissions are evaluated on the NST and not at the SQL Server level. These Role permissions are cached to ensure good performance.

Hence, there is no need to **Synchronize All Logins**, since Users are no longer created as SQL Server logins with their associated SQL Users. In addition, there is no requirement to set up delegation for three tier **shared nothing** environments because the NST does not need to impersonate the end user of the client session.

The security requirements for the SharePoint and Web Clients are more complex and are covered elsewhere.

Transaction Isolation Levels

The following are the Microsoft Dynamics NAV transaction types:

- **UpdateNoLock**
  - If the current connection has no other locks on the relevant table then the NST will perform **dirty reads,** when using this transaction type, i.e., it will read the rows from the table regardless of any locking applied by other connections.
  - If the current connection already holds locks for the relevant table then this transaction type will cause the NST to use the UPDLOCK table hint when reading records from the relevant table.
- **Update and Snapshot**
  - If the current connection has no other locks on the relevant table then the NST will use the RepeatableRead transaction isolation level.
  - If the current connection already holds locks for the relevant table then this transaction type will cause the NST to use the UPDLOCK table hint when reading records from the relevant table.
- **Browse and Report**—Will always use **dirty reads**.

In Microsoft Dynamics NAV 2013, the default transaction isolation level is now REPEATABLE READ, changed from SERIALIZABLE in previous versions of Microsoft Dynamics NAV. With this transaction isolation level, only committed data is read because shared locks are placed and held until the transaction completes.

No other transaction can modify read records until transaction completes. However, **Phantom reads** are now theoretically possible because range locks are not used with REPEATABLE READ. Therefore, another transaction could in theory insert rows within the filter criteria.

Note:

Note that using REPEATABLE READ rather than SERIALIZABLE affects data access functions, such as FIND, FINDSET, NEXT, etc., when the transaction type is set to Update or Snapshot. Over cautious range, locking problems have been solved at the expense of potential (albeit unlikely) phantom reads. Blocking is reduced, since individual records as opposed to ranges are locked

The default Read type will still use READ UNCOMMITTED isolation level (i.e., dirty reads).

More:

For more information, see:

http://msdn.microsoft.com/en-us/library/ms191272(v=sql.105).aspx

Smarter SQL Statements

In Microsoft Dynamics NAV 2013, fewer SQL statements are issued in cases where you filter on FlowFields. Whenever possible, the NST expands the issued SQL query (the same statement that contains the filter expression) to include the FlowField value.

In previous versions of Microsoft Dynamics NAV, a SQL statement was executed for each FlowField in the filter expression per record in the main table in order to calculate the FlowField value in the filter.

Hence, on average, far fewer SQL statements are executed in Microsoft Dynamics NAV 2013, thus saving many roundtrip to SQL Server.

Exceptions, where the NST reverts to old behavior:

- When the **ValueIsFilter** property is enabled within the FlowField definition.

  This is a flag on the **Table Filter** part of the CalcFormula, which determines how the system interprets the contents of the field referred to in the Value column in the Table Filter window.

  For example, if the field contains the value 1000..2000, setting the ValueIsFilter option will cause this value to be interpreted as a filter rather than as a specific value.

- FlowField definition has multiple filter expressions with the same field included. For example:

```
FlowField1 = SUM(BaseTable.SomeValue)
WHERE CurrTable.SourceNo=BaseTable.MyNo,
CurrTable.SourceNo=BaseTable.MyAlternateNo
```

  e.g. depending on whether MyAlternateNo is blank, the criteria will be:

  WHERE SourceNo=MyNo or SourceNo=MyAlternateNo.

In Microsoft Dynamics NAV 2013, a single statement is normally used when requesting a result set (using FIND, FINDSET, etc.), which is filtered on marked records.

The only exception would be, if so many individual marks are in place that SQL statement limitations are reached in which case multiple queries will be issued.

In Microsoft Dynamics NAV 2009 R2, the NST issues a SQL statement for every mark which is in place. Besides slower performance due to the extra round trips to the server, the previous behavior could also result in an inconsistent sort order because the intermediate results were buffered and eventually sorted in a temporary table.

Calculated Fields

For Microsoft Dynamics NAV 2013, CALCSUM and CALCFIELDS execution is decoupled from Microsoft Dynamics NAV SIFT index definitions. This means that if any of the conditions for

using SIFT indexes are not true, then Microsoft Dynamics NAV traverses all records in the base table to perform the calculation instead of using SIFT.

This can reduce the number of required SIFT indexes, which can improve performance. In earlier versions of Microsoft Dynamics NAV, if the conditions for using SIFT indexes were not true and the **MaintainSIFTIndex** property was enabled, then you receive an error when you call the CALCSUM or CALCFIELDS function. This provided a degree of protection in earlier versions against accidentally requesting a sorting for which no index existed.

In Microsoft Dynamics NAV 2013, an index is not required to support a certain sorting, but sorting without an index could lead to bad performance if a search returns a large result set, which would then have to be sorted before the first row is returned.

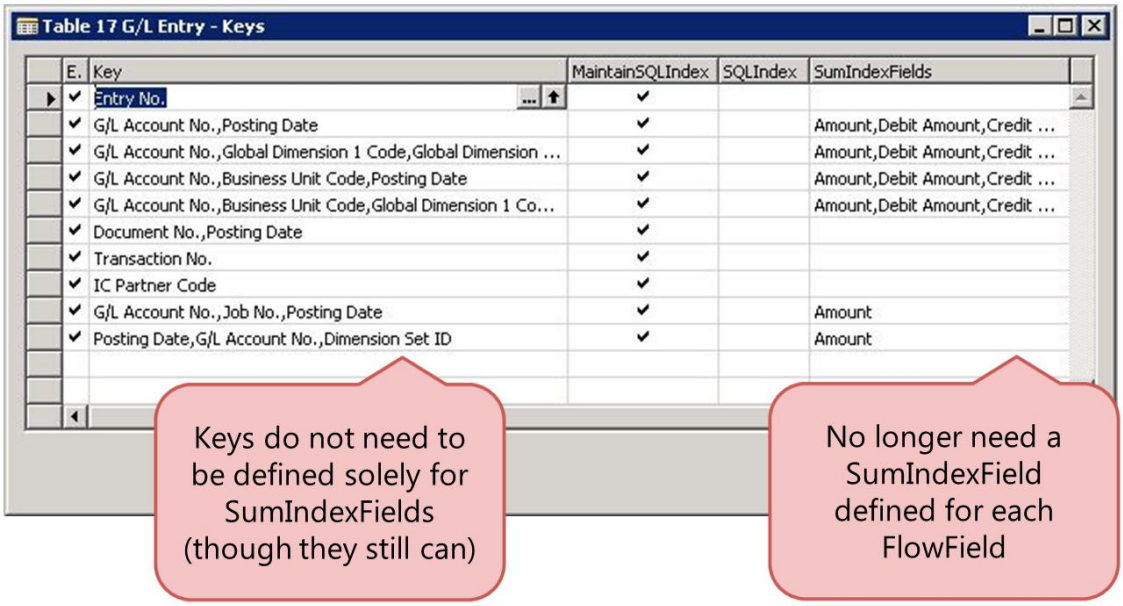

Figure 7: SIFTIndexFields Defined on Keys

While it is no longer required to create SIFT when summing or calculating it may still be a good idea depending on the likely size of the data which will be handled by the CALCSUM or CALCFIELDS functions.

If you have a SIFT defined in the relevant SumIndexFields it will incur an overhead every time the table is updated as the SIFT index must be maintained in SQL Server. However, it will speed up READ access to the relevant aggregated value if there are many records involved in summarizing the aggregated value.

So, when deciding whether or not to create and maintain a SIFT index, consider the following two factors:

- How heavily updated is the table which contains the raw values to be aggregated. The more updates expected on this table the less worthwhile it is to maintain a SIFT index.

- When accessing the relevant aggregated value, how many records will be summarized.

  If a typical query for the aggregated value would only be summarizing a small number of records then you would not gain much efficiency by creating a SIFT.

  However, if the number of records involved would be several thousand then the performance may be improved for READ access by creating a SIFT.

- How frequently will the aggregated value be required? If the aggregated value in question is accessed rarely then maintaining a SIFT index may be less useful.

In Microsoft Dynamics NAV 2013, the behavior when using CALCFIELDS with a field of type BLOB has changed.

In previous versions of Microsoft Dynamics NAV, if you wrote to a BLOB OutStream but did not insert or modify the record in the database, and then called the CALCFIELDS function on the BLOB field, you would get the value of the BLOB based on what you wrote to the OutStream, not based on what was currently in the database.

In Microsoft Dynamics NAV 2013 in the same scenario, you get the value of the BLOB that is in the database. Similarly, in Microsoft Dynamics NAV 2013, if you call the CALCFIELDS function on a new record that has not been inserted into the database, then you clear the BLOB field from the record.

When you call CALCFIELDS on a BLOB field, which is empty it will return BLANK.

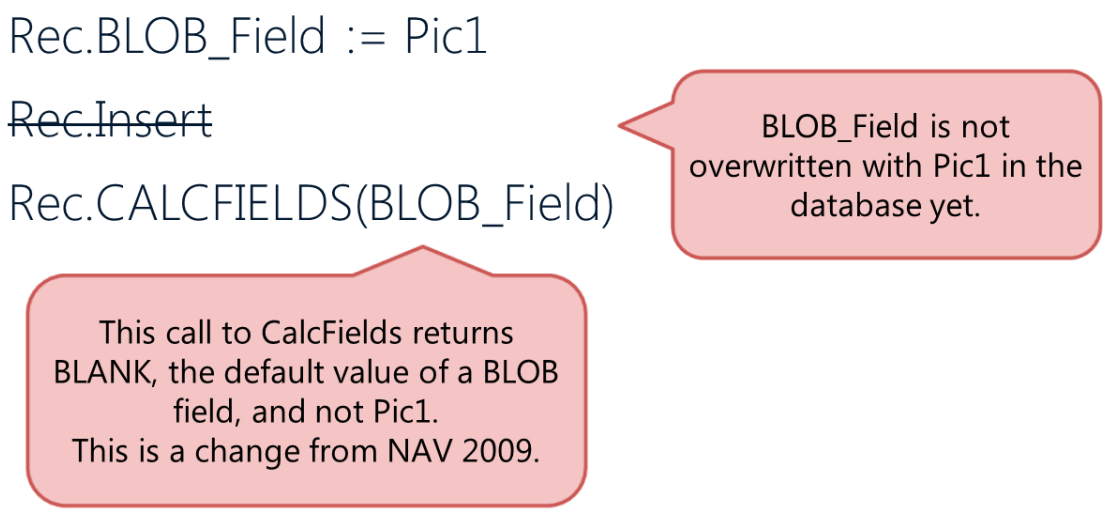

Figure 8: How CALCFIELDS with BLOBs has Changed

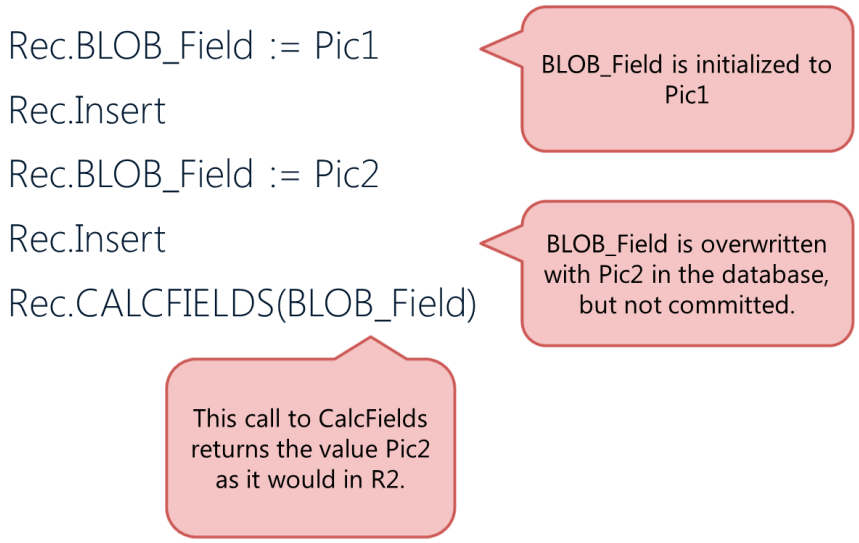

Figure 9: CALCFIELDS with BLOB Expected Usage

Modifying an Old Record after Commit

In Microsoft Dynamics NAV 2013, it is not possible to modify an old record once it has been committed, without explicitly reading it again from the database (i.e., GET call).

For example, consider the following code sample:

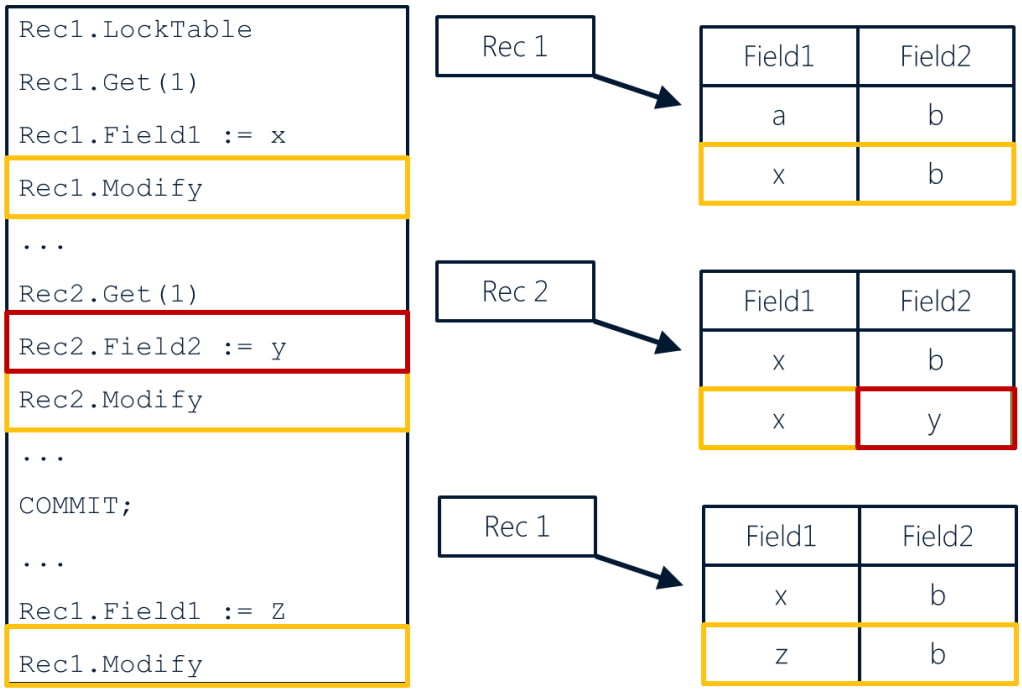

Figure 10: Inappropriate Code

In the previous code sample, Rec1 and Rec2 are Record variables defined on the same table. On the right side of the diagram, you can see the record values before and after each MODIFY statement.

Note: The final MODIFY statement was executed on Rec1 without having re-read the record since the COMMIT statement

In Microsoft Dynamics NAV 2009, this code would run without any error being generated. However, unexpected results would have been produced, i.e., Field2 was set to the value **y** by the second MODIFY mentioned previously but this value has been lost.

If you run the same code in Microsoft Dynamics NAV 2013, you would encounter the following runtime error message. This error is an enhancement which will avoid unexpected results with the data as described previously:

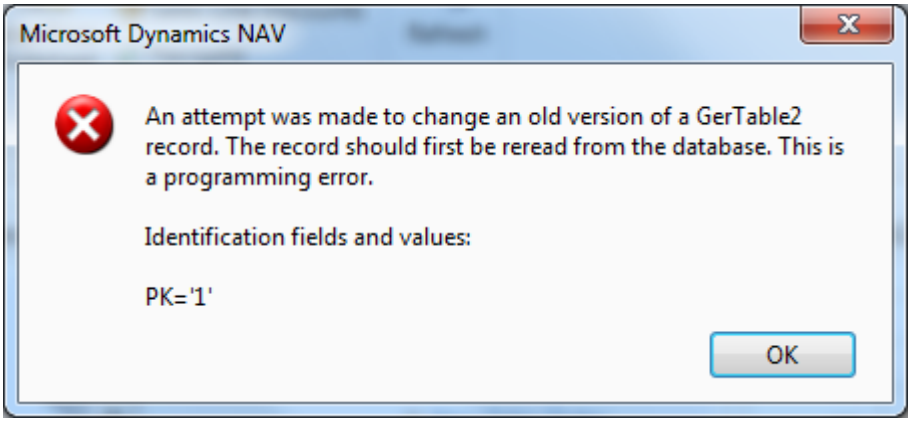

Figure 11: Microsoft Dynamics NAV 2013 Sample Error

If the code is amended to add the line indicated in the following, it will work as expected all versions of Microsoft Dynamics NAV:

Code Segment 1: Code Correction

```
Rec1.LOCKTABLE;
Rec1.GET(1);

Rec1.MyFirstField := 'x';
Rec1.MODIFY;

Rec2.GET(1);
Rec2.My2ndField := 'y';
Rec2.MODIFY;

COMMIT;

Rec1.GET(1);              //ADD a GET after the COMMIT
Rec1.MyFirstField := 'Z';
```

```
Rec1.MODIFY;
```

Sort Order of Temporary Tables

The sort order of physical tables within Microsoft Dynamics NAV is defined by the collation in SQL Server for the database. However, the sort order for Microsoft Dynamics NAV temporary tables (not to be confused with SQL Server temporary tables) in previous releases of Microsoft Dynamics NAV 2009 was based on the native Microsoft Dynamics NAV sorting algorithms, which did ordinal sorting.

In Microsoft Dynamics NAV 2013, the Microsoft Dynamics NAV temporary table implementation has been re-written and is now Managed code. As part of this process the sorting of tables has been made consistent for Microsoft Dynamics NAV temporary tables and physical SQL Server tables.

Comparisons for data in Microsoft Dynamics NAV temporary tables use the Windows API and thus use the Windows collation, which is linked to the relevant collation on SQL Server, to define sorting rules. Physical SQL Server tables also use the specified Windows collation in the database to define sorting rules. This may cause a change in behavior for application code written in previous versions of Microsoft Dynamics NAV, which depends on the old sort order for Microsoft Dynamics NAV temporary tables. These scenarios will need to be reviewed and revised as part of any upgrade plan.

SetTable Function

In Microsoft Dynamics NAV 2013, the RecRef.SETTABLE(ToRecord) function has been modified to ensure consistency in the items being copied between RecordRef and Record variables.

The SETTABLE function copies the values of the record in the RecordRef variable to its argument, i.e., the ToRecord record. It also transfers the current key, any filters applied, and the current filter group from the record in the RecordRef to the record in the argument.

The behavior described for SETTABLE above varies slightly from earlier versions of Microsoft Dynamics NAV. Specifically, in Microsoft Dynamics NAV 2009 R2, the current key, any filters applied, and the current filter group were copied in the opposite direction from the record in the argument (ToRecord) to the record in the RecordRef variable to remain consistent with the classic runtime. Therefore, Microsoft Dynamics NAV 2013 has been designed to ensure a consistent and intuitive copy direction.

SetAutoCalcFields (New in Microsoft Dynamics NAV 2013)

In previous versions of Microsoft Dynamics NAV CALCSUMS and CALCFIELDS were used to calculate FlowField value of a record after it was retrieved from the database, thus requiring an extra query to the SQL Server to retrieve the FlowField value.

However, sometimes it is known before querying the database whether the FlowField value needs to be retrieved alongside the record.

In Microsoft Dynamics NAV 2013, the Record API has been expanded to allow for the simultaneous retrieval of the record and calculation of the FlowField. This new approach is similar to using CALCSUMS and CALCFIELDS in that they all calculate the FlowField value.

However, how and when the calculation is done is different as can be seen in the following Code Segments:

Code Segment 2: Example using CalcFields

```
IF (customers.FINDSET) THEN
BEGIN
  REPEAT
  customers.calcFields("Sales (LCY)");
  IF (customers."Sales (LCY)" <> 0) THEN
    BEGIN
      customers.calcFields("Balance (LCY)");
      IF (((customers."Balance (LCY)" / customers."Sales (LCY)") * 100) > 40)
THEN
        BEGIN
          customers.LOCKTABLE;
          customers.Blocked := customers.Blocked::Invoice;
          customers.MODIFY;
        END;
    END;
  UNTIL (customers.NEXT()=0);
END;
```

The code shows that the CalcFields() function is executed twice.

- In the first case, for each record in the set of customers, another query must be sent to SQL to retrieve this FlowField value.
- In the second case, for each record in the set of customers whose sales is not zero, another query must be sent to SQL to retrieve the second FlowField value.

For such a scenario, it would be more efficient to retrieve the FlowField values along with the rest of the record.

Microsoft Dynamics NAV 2013 introduces the SetAutoCalcFields function, which allows you to mark ,which FlowFields should automatically be calculated as part of data retrieval.

The function signature is:

```
[OK :=] Record.SETAUTOCALCFIELDS([Field1,Field2, …])
```

Where,

Record is the Record for which you want a FlowField calculated.

Field1, Field2, etc. are the FlowFields that you want calculated.

OK is the Boolean value specifying whether the tagging was set successfully.

Utilizing the SetAutoCalcFields function significantly reduces the number of SQL calls to retrieve a group of records, plus their corresponding FlowField(s) values.

Instead of having one SQL Server query to retrieve the dataset and another query to calculate the FlowFields on each row, we can now have a single query which retrieves the entire dataset and desired FlowFields. This gives C/AL developers a good opportunity to optimize existing code.

In Microsoft Dynamics NAV 2013, a Record variable on a Page will automatically take advantage of this function implicitly.

Code Segment 3: Example using SetAutoCalcFields

```
customers.SETAUTOCALCFIELDS("Sales (LCY)", "Balance (LCY)");

IF (customers.FINDSET) THEN
BEGIN
  REPEAT
  IF (customers."Sales (LCY)" <> 0) THEN
    BEGIN
      IF (((customers."Balance (LCY)" / customers."Sales (LCY)") * 100) > 40)
THEN
        BEGIN
          customers.LOCKTABLE;
          customers.Blocked := customers.Blocked::Invoice;
          customers.MODIFY;
        END;
    END;
  UNTIL (customers.NEXT()=0);
END;
```

The code will produce the same results as the earlier Code Segment 2. However, because of the use of the SetAutoCalcFields function, it will be more efficient as all FlowField values will be returned with the record set thus avoiding the need for extra round trips to the SQL Server.