CHAPTER 11: OPTIMIZING SQL SERVER

Objectives

The objectives are:

- Discuss the database options available in Microsoft Dynamics™ NAV 5.0.
- Explain how tables and indexes work and how they are stored.
- Discuss collation options and descriptions.
- Discuss Windows locale.
- Discuss backup options.
- Explain how the SQL Server Query Optimizer works.
- Discuss the NDBCS driver.
- Discuss the optimization of database cursors.
- Explain the performance impact of locking, blocking, and deadlocks.
- Explain how SIFT data is stored.

Introduction

There are two server options to choose from when using Microsoft Dynamics NAV: the “classic” or “native” Microsoft Dynamics NAV server, and Microsoft SQL Server. Each has advantages and disadvantages. This section covers the differences between the two, but SQL Server is the only database option for installations that use the Microsoft Dynamics Client.

Microsoft Dynamics NAV Database Server

The Microsoft Dynamics NAV database server can be characterized as an Indexed Sequential Access Method (ISAM) server. The data is stored in B-TREE structures, with the primary key used to store the data physically on disk and secondary keys used to find a range of records and point to the data location based on the primary key. The Microsoft Dynamics NAV Server does not provide advanced data retrieval mechanisms -- you have to specify which index is to be used, or the data will be retrieved by scanning the primary key. There is no read-ahead mechanism (apart from the backup system). Records are sent to the client one by one, while commands sent to the server are sent in batches to reduce network traffic. There are a very limited number of commands executed at the server level (MODIFYALL for example), so virtually all the data manipulation is done at the client level.

The Microsoft Dynamics NAV Server supports explicit and implicit record locking. Implicit locking takes place when a modification of a record is performed on a table, at which time Microsoft Dynamics NAV Server issues a table-lock, which it holds until the transaction is complete. This sometimes compromises concurrency.

However, there are two important and advantageous features in Microsoft Dynamics NAV Server: SumIndexField Technology (SIFT) and Data Versioning.

SIFT is designed to improve performance when carrying out activities like calculating customer balances. In traditional database systems this operation involves a series of database calls and calculations. SIFT makes calculating sums for numeric table columns extremely fast, even in tables that contain thousands of records.

Data versioning is designed to insure that reading operations can be performed in a consistent manner. For example, when you print Balance Sheet, it will balance without the need to issue locks on the G/L Entry table. Conceptually this works as though the server creates a snapshot of the data. During a data update, the modified data is kept private until a commit appears, and then the new data is made public -- with a new version number. Data versioning allows optimistic concurrency, which can improve performance.

SQL Server

SQL Server is a comprehensive database platform providing 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 retrieving 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 accomplished with an NBDCS driver, which is discussed later in this chapter.

When SQL Server receives a query (in the form of a T-SQL statement), it uses the SQL Query Optimizer to create and execute the query. For example, Query Optimizer decides which index to use, if it should use parallel execution, and so forth. Query Optimizer assumes that the client generates unoptimized queries and therefore it creates and executes queries according to its own logic. The primary criteria that Query Optimizer uses to decide which execution plan to use is the performance cost of executing the query.

Like the Microsoft Dynamics NAV Server, SQL Server stores data in B-TREE structures. The primary index is used to store the data physically on disk, and secondary indexes are used to find a range and point to the data in the primary index. SQL Server differs from the Microsoft Dynamics NAV Server. The former is called clustered index, and the latter non-clustered index.

Unlike the Microsoft Dynamics NAV Server, SQL Server can do server-side processing, but – for the most part – Microsoft Dynamics NAV does not use this ability. Microsoft Dynamics NAV uses the same strategy for database access as it does with the Microsoft Dynamics NAV Server, so a limited number of commands are executed on a server-side basis (such as DELETEALL).

Both Microsoft Dynamics NAV clients retrieve data record by record, but SQL Server is set-based, so it doesn't provide a fast way to do this. Therefore SQL Server uses cursors for record-level access. However, cursors are very expensive compared to set retrievals. Later, we will discuss how to reduce this overhead where possible.

SIFT is supported by SQL Server by using extra summary tables that are maintained through table triggers. When an update is performed on a table containing SIFT indexes, a series of additional updates are necessary to update the summary tables. This also imposes an extra performance penalty – one that grows as the number of SIFT indexes on a table increases.

SQL Server supports data versioning, but it is simulated. Microsoft Dynamics NAV Server adds a timestamp column to every regular table and reads data using the timestamp and the READUNCOMMITED isolation level. Therefore when you modify a record, it will re-read the data with UPDLOCK and modify the data with a filter on the original timestamp. This prevents data that has been read "dirty" from being stored. The downside is that this method allows users to see data processed by other clients. If you want to prevent this, you have to issue a lock – this will ensure that the data is read with the READCOMMITED isolation level and also will guarantee that no other user can modify the same set of data that you read (and locked). The downside of locking is reduced concurrency, as some users may be blocked for the duration of your activity lock.

On the other hand, SQL Server provides much more sophisticated locking options compared to Microsoft Dynamics NAV Server, including record-level locking, which vastly improves the efficiency of parallel operations when performed on the same set of tables.

Representation of Microsoft Dynamics NAV Tables and Indexes in SQL Server

Microsoft Dynamics NAV tables can store data unique to each company, or data common to all companies. Developers can set the DataPerCompany table property to determine this. When you create a table with the Microsoft Dynamics NAV table designer, a corresponding table in SQL Server is created in the Microsoft Dynamics NAV database with a name in the following format:

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

Microsoft Dynamics NAV uses naming conventions complying with SQL Server, such as not using special characters. If the table is common to all companies, then the name will be "G_L Entry". The table will have a number of indexes, representing the indexes that have been designed and enabled in the Microsoft Dynamics NAV 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

Microsoft Dynamics NAV also clusters the primary key index by default. Also, by default, Microsoft Dynamics NAV will add the remainder of the primary key to every secondary key, making the indexes unique. This conforms to the "best practices" defined for SQL Server. Developers can make additional changes to the way indexes are defined on SQL Server using the MaintainSQLIndex, SQLIndex, and Cluster properties on the indexes defined in the table. To obtain a list of indexes and their definition in SQL Server, run the sp_helpindex stored procedure, as follows:

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

The query outputs the index name, if the index is clustered or unique, whether or not there is a primary key constraint, and also the index keys defined in the index.

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

SQL Server Terminology	Microsoft Dynamics NAV Terminology
Primary key constraint	Primary key
Clustered index	No equivalent, same as primary key
Non-clustered index	Secondary key
Index key	Field in a key definition

SQL Server allows non-unique indexes; however, because SQL Server makes the indexes unique internally by adding extra index keys, it is a good practice to make them unique to begin with. Additionally, SQL Server allows a table not to have a clustered index. This is called a "heap" and can be used for archiving, because data is stored as it comes. Heaps are not recommended for tables that are read, though, because reading from an unstructured source is rather slow.

Collation Options

SQL Server supports several collations. A collation encodes the rules governing the proper use of characters for either a language, such as Macedonian or Polish, or an alphabet, such as Latin1_General (the Latin alphabet used by Western European languages). Each SQL Server collation specifies 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 characters are evaluated in comparison operations.
- The sort order to use for non-Unicode character data types (char, varchar, and text).
- The code page used to store non-Unicode character data.

SQL Server collations can be specified at any level. When developers install an instance of SQL Server, they specify the default collation for that instance. Each time they create a database, they can specify the default collation used for the database. If they do not specify a collation, the default collation for the database is the default collation for the instance. Whenever they define a character column, variable, or parameter, they can specify the collation of the object. If they do not specify a collation, the object is created with the default collation of the database.

If all of the users of their instance of SQL Server speak the same language, they should pick the collation that supports that language. For example, if all of the users speak French, they should choose the French collation. If the users of their instance of SQL Server speak multiple languages, they should pick a collation that best supports the requirements of the various languages. For example, if the users generally speak western European languages, they should choose the Latin1_General collation.

Developers can use the Microsoft Dynamics NAV client to change the collation settings when they create the database, or afterward with some limitations. Developers should use the Windows Locale option to match collation settings in instances of SQL Server 2000 and above. Developers should use SQL Collations to match settings that are compatible with the sort orders in earlier versions of SQL Server.

Windows Locale

Developers should change the default settings for Windows Locale (that is Windows collation) only if users' installation of SQL Server must match the collation settings used by another instance of SQL Server, or must match the Windows Locale of another computer.

Collation Description

Select the name of a specific Windows collation from the drop down list. Note that if users put a checkmark in the Validate Code Page field, 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 selected - these are Binary, Case-sensitive and Accent-sensitive. Binary is the fastest sorting order, and is case-sensitive. If Binary is selected, the Case-sensitive and Accent-sensitive options are not available.

SQL Collations

The SQL Collations option is used for compatibility with earlier versions of Microsoft SQL Server. Select this option to match settings compatible with SQL Server version 7.0, SQL Server version 6.5, or earlier.

SQL Server Replication

SQL Server replication enables database administrators to distribute data to various servers throughout an organization. This is useful for a number of reasons, such as:

- Load balancing: Replication enables users to disseminate data to a number of servers and then distribute the query load among those servers.

- Offline processing: Users can manipulate data from their database on a machine that is not always connected to the network.
- Redundancy: Replication enables developers to build a fail-over database server that is ready to pick up the processing load at a moment's notice.

In any replication scenario, there are two main components:

- Publishers have data to offer to other servers. Any given replication scheme may have one or more publishers.
- Subscribers are database servers that are set to receive updates from the publisher when data is modified.

There is nothing preventing a single system from acting in both of these capacities. In fact, this is often done in large-scale distributed database systems. Microsoft SQL Server supports the following three types of database replication:

- Snapshot replication acts in the manner its name implies. The publisher simply takes a snapshot of the entire replicated database and shares it with the subscribers. Of course, this is a very time- and resource-intensive process. For this reason, most administrators do not use snapshot replication on a recurring basis for databases that change frequently. There are two scenarios where snapshot replication is commonly used. First, it is used for databases that rarely change. Second, it is used to set a baseline to establish replication between systems while future updates are propagated using transactional or merge replication, the other two types of database replication.
- Transactional replication offers a more flexible solution for databases that change on a regular basis. With transactional replication, the replication agent monitors the publisher for changes to the database and transmits those changes to the subscribers. This transmission can take place immediately or on a periodic basis.
- Merge replication allows the publisher and subscriber to independently make changes to the database. Both entities can work without an active network connection. When they are reconnected, the merge replication agent checks for changes on both sets of data and modifies each database accordingly. If changes conflict with each other, it uses a predefined conflict resolution algorithm to determine the appropriate data. Merge replication is commonly used by laptop users and others who cannot be constantly connected to the publisher.

Microsoft Dynamics NAV Database Replication

Microsoft Dynamics NAV is a transactional database, and there is one situation that SQL Server replication cannot address. This is having multiple sites without reliable connections update data at any time with any frequency and maintain transactional integrity. The problem is that as sites become disconnected, transaction integrity is rapidly lost. Users on a disconnected site could make a change to, or insert a record with the same primary key as, a record that is deleted by users on another site. When the site reconnects, the update fails because the record no longer exists on the master database. As a result, merge replication should not be used for installations across multiple sites, but it can be used for some specific cases, such as merging data from a third-party application into some integration tables in Microsoft Dynamics NAV.

Another issue that affects replicated databases is that under some circumstances, the replication mechanism changes the TimeStamp column in the Microsoft Dynamics NAV tables to GUIDs on the replicated tables, making the database unusable by Microsoft Dynamics NAV. Additionally, developers must not use replication on SIFT tables, as these are maintained by table triggers. Therefore, developers should consider log shipping before committing to a replication strategy for resilience reasons. If they do use replication, they should make sure that it is properly tested for functionality and performance impact. There is more overhead if developers run replication on their server, so the subsystems must be sized accordingly.

To summarize, replication should be used sparingly with Microsoft Dynamics NAV databases, and only if there is no other mechanism to distribute and/or share data, such as log shipping, using views, using DTS/SSIS (SQL Server Integration Services), dataports, and so on.

Backup Options

Developers setting up an SQL Server for a company with a Microsoft Dynamics NAV installation need to ensure that the data is backed up regularly in case data is lost and disaster recovery (DR) needs to be executed. This can happen, for example, if your disk subsystem fails to read from a corrupted disk, or the entire server was lost during a fire or similar disaster. DR strategies vary depending on the level of the customer needs, but also on the budget available, as better DR generally costs more. All DR relies on online and offline resiliency. For example, with RAID1 (disk mirroring), the data is written to two disks in case one fails, or with clustered servers, you can have one active to do the work and one passive which can be used if the primary server fails.

In case data is lost, you need to restore a backup. There are various backup options with Microsoft Dynamics NAV on SQL Server. While the Microsoft Dynamics NAV client side backup option still exists (manually doing a backup through Tools, Backup), there are more elegant tools in SQL Server. The Microsoft Dynamics NAV backup and restore functionality remains for migration tasks such as users wanting to move data from one server with one collation into a different server with non-compatible collation, or for example if users want to use multiple database files and spread the data into those while restoring the Microsoft Dynamics NAV backup. Note that restoring the Microsoft Dynamics NAV backup is fully logged and therefore requires a massive amount of space in the log file.

SQL Server itself offers a number of options for disaster recovery. To start with, users can choose what recovery model the database will be using, either Simple or Full. Bulk-Logged is another option, but it is rarely used and is similar to Full anyway, because only bulk operations logs are simplified. After choosing the recovery model, developers start planning their backup strategies using full, differential, and log backups, or a combination of those.

Simple Recovery Model

The simple recovery model is using the log file just to record open transactions. After committing the transaction to the database, the log space is reclaimed. The benefit is that the log file can be quite small and simpler, and therefore faster. The disadvantage is that in the case of a disaster, users can only recover transactions to the end of the previous backup. Then they have to redo all transactions again. When using Simple Recovery, the backup interval should be long enough to keep the backup overhead from affecting production work, yet short enough to prevent the loss of significant amounts of data.

Full Recovery Model

The Full Recovery and Bulk-Logged Recovery models provide the greatest protection for data. These models rely on the transaction log to provide full recoverability and to prevent work loss in the broadest range of failure scenarios. The disadvantage is the size of the log file and amount of data logged; the advantage is that if users have a disaster, they can recover to any point of time, provided that the log itself has not been damaged.

Full Backup and Restore

Full database backup backs up the entire database, including the transaction log. When users restore this backup, they have a fully operational database from the time of the backup. This type of backup requires the most space.

Differential Backup and Restore

A differential backup creates a copy of all the pages in a database modified after the last database backup. Differential logs are used primarily in heavily used systems where a failed database must be brought back online quickly. Differential backups are smaller than full database backups; therefore, they have less of an effect on the system while they run.

Transaction Log Backup and Restore

A transaction log backup makes a copy of only the log file. A log file backup by itself cannot be used to restore a database. A log file is used after a database restore to recover the database to the point of the original failure. Users cannot use transaction log backup and restore for databases using the Simple recovery model because the log file contains only uncommitted transactions.

Reducing Recovery Time

Using full database backup, differential database backup, and transaction log backup together can reduce the amount of time it takes to restore a database back to any point in time after the database backup was created. Additionally, creating both differential database and transaction log backups can increase the robustness of a backup in the event that either a transaction log backup or differential database backup becomes unavailable, such as due to media failure.

Typical backup procedures using database, differential database, and transaction log backups create database backups at longer intervals, differential database backups at medium intervals, and transaction log backups at shorter intervals. An example would be to create database backups weekly, differential database backups one or more times per day, and transaction log backups every ten minutes.

If a database needs to be recovered to the point of failure, such as due to a system failure, perform the following steps:

1. Back up the currently active transaction log. This operation will fail if the transaction log has been damaged.
2. Restore the last database backup created.
3. Restore the last differential backup created since the database backup was created.
4. Apply all transaction log backups, in sequence, created after the last differential backup was created, finishing with the transaction log backup created in Step 1.

Note that if the active transaction log cannot be backed up, it is possible to restore the database only to the point when the last transaction log backup was created. Changes made to the database since the last transaction log backup are lost and must be redone manually.

By using differential database and transaction log backups together to restore a database to the point of failure, the time taken to restore a database is reduced, because only the transaction log backups created since the last differential database backup was created need to be applied. If a differential database backup was not created, then all the transaction log backups created since the database was backed up need to be applied.

Backup Impact on Performance

Systems with the highest resilience running frequent backups and Full recovery model will incur much higher overhead compared to a system with low resilience using Simple recovery model and infrequent backups in which all the machine power is dedicated to executing all the transactions. Developers should discuss with the customer the different aspects and implement the best compromise, balancing risk with performance.

SQL Server Query Optimizer

Query Optimizer is the heart of SQL Server when making a decision on how to execute a query. SQL Server collects statistics about individual columns (single-column statistics) or sets of columns (multi-column statistics). Statistics are used by the query optimizer to estimate the selectivity of expressions, and thus the size of intermediate and final query results. Good statistics allow the optimizer to accurately assess the cost of different query plans, and choose 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) kept in an internal-only table.

SQL Server maintains some information at the table level. These are not part of a statistics object, but SQL Server uses them in some cases during query cost estimation. This data is stored at the table level:

- Number of rows in the table or index (rows column in sys.sysindexes)
- Number of pages 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 the statistics were collected
- Number of rows used to produce the histogram and density information (described hereafter)
- Average key length
- Single-column histogram, including 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. In general, these intervals are of nonequal size.

Users can see 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:

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

The result set will have three sections, similar to those below:

Updated	Rows	Rows Sampled	Steps	Density	Average key length
Jun 14 2006 2:51PM	26046	26046	3	0.0	26.237772

All density	Average Length	Columns
0.33333334	4.0	Document Type
1.0416667E-2	11.866083	Document Type, Customer No_
5.165289E-4	19.866083	Document Type, Customer No_, Posting Date

RANGE_HI_KEY	RANGE_ROWS	EQ_ROWS	DISTINCT_RANGE_ROWS	AVG_RANGE_ROWS
1	0.0	23689.0	0	0.0
2	0.0	10.0	0	0.0
3	0.0	2341.0	0	0.0
4	0.0	6.0	0	0.0

Imagine that a user is filtering on "Document Type" column in the "Cust. Ledger Entry" table, for example looking for all "Credit Memo" type entries. The "Credit Memo" type entries have a value of "Document Type" equal to 3. Microsoft Dynamics NAV will issue a query similar to this:

```
SELECT * FROM
  "CRONUS International Ltd_$Cust_ Ledger Entry"
WHERE
  "Document Type" = 3
GO
```

The query optimizer will have to analyze usefulness of every index in the table so that the query is executed at minimal cost, minimizing first the cost of data retrieval, followed by costs of sorting, and so on. When analyzing this particular index (index $6) from data retrieval aspect, the query optimizer will make the majority of its decisions based on the statistics in the following way. "Document Type" is filtered, but there are only three distinct values in the index (refer to the "All density" column in the preceding table) indicating that 0.3334 of the table is within the filtered set. Additionally, since the histogram exists on the "Document Type,"it will look up the information (RANGE_HI_KEY = 3) and see that there are about 2341 rows in the set, indicating that 0.0898 of the table is within the filtered set (2341 rows out of total 26046 rows). Based on this, the query optimizer will decide that there is no point in using this index to do this operation because it would have to load the index, scan the range, and look up the data in the clustered indexes to return the results. Since there is no other better way to read the data, it will decide to do Clustered Index Scan instead. As a rule of thumb, if the selectivity is close and better than 1%, the index will be considered beneficial. Be careful with this simple rule, though, because operations such as SELECT TOP 1 (asking for the first record in a set) will escalate the index benefit, and the index will probably be used.

Continuing this example, if users filtered on a more unique value, making the index help with selectivity, such as:

```
SELECT * FROM
  "CRONUS International Ltd_$Cust_ Ledger Entry"
WHERE
  "Document Type" = 4
GO
```

The query optimizer will know – in this example -- that only 6 out of 26046 records qualify (refer to the previous histogram or RANGE_HI_KEY = 4) and will make use of the index. It will seek in the non-clustered index followed by the bookmark lookup to the clustered index to get the relevant records.

```
SELECT * FROM
  "CRONUS International Ltd_$Cust_ Ledger Entry"
WHERE
  "Document Type" = 4
GO
```

If users further filter on the next index key, like this:

```
SELECT * FROM
  "CRONUS International Ltd_$Cust_ Ledger Entry"
WHERE
  "Document Type" = 3 AND
  "Customer No_" = '10000'
GO
```

The combined selectivity will be used, and a plan calculated, and 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 using OR operator, among others, there is no way that SQL Server can combine the subqueries, and may result in bad behavior. For example:

```
SELECT * FROM
  "CRONUS International Ltd_$Cust_ Ledger Entry"
WHERE
  "Customer No_" = '10000' AND
  (("Document Type" = 2) OR ("Document Type" = 4))
GO
```

In this example, the query optimizer will decide to use the index, doing non-clustered index seek, but will have to traverse the area of "Document Type" = 3, because the set is read from beginning to end, which will often have a similar effect, as if user did a table scan.

Similarly, if users leave one of the index keys unfiltered, all the sub-tree index entries have to be scanned. There is one simple rule with regards to performance: scan is bad, seek is good. Developers need to avoid scans as much as they can, making sure 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 overly complex and designed to fully match the entire query, for example with 8 index keys. It might be that if the index has only 4 index keys, SQL Server will have to scan a slightly larger set to provide the required several records, but at the extreme cost of having to maintain the composite index, delaying every modification in the table, because SQL Server has to update the index accordingly. In a majority of cases, users have to optimize the transaction speed, and if users over-index the tables, then users have to pay a heavy price. It might be that a specific report is slow when fewer composite indexes are used, but it is worth it, since processing (such as posting inventory) will be quicker.

Additionally, if a set is to be 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 above demonstrates that badly designed and badly used indexes can severely impact SQL Server performance. Adhere to the following principles:

- Minimize the number of indexes for faster table updates.
- Design indexes with index keys of good selectivity.
- Put index keys with higher selectivity toward the beginning of an index.
- Put index keys that are more likely to be filtered toward the beginning of an index.

- If the filtered index keys point to approximately 50 records, there is no need to add extra index keys to improve the index selectivity; SQL Server will return the right set.
- If the filtered index keys point to approximately 50 records, there is no need to add extra index keys to support sorting; SQL Server will return the set sorted as desired.
- It is better to use more composite indexes than a lot of simple indexes; in other words, it is better to "combine" use rather than have many "specific" indexes.
- There is no point of indexing "empty" (not used) columns , since that just creates an extra overhead for no benefit.
- Make sure that users filter on unique values in indexes; otherwise it will perform similar to table scans.
- Do not make "overselective" index keys. If users index on DateTime fields, for example, they will have a unique index leaf in the index for each record in the table.
- Put date fields toward the end of the index, as this is not always filtered. But if an index is always filtered on a unique value, then it is a good index.

Finally, if users doubt what index is good, imagine a telephone book or list of personal details and they are trying to find people by Name and Surname, or Date of Birth, or Security Number, and so on. compared to Gender for example. Try to use similarities in that and apply common sense.

Optimizing a Microsoft Dynamics NAV Application

There are a number of areas where users should focus when optimizing Microsoft Dynamics NAV applications. These areas are, in order of importance (based on the processing costs):

- SIFT
- Indexes
- Cursors
- Locks
- Suboptimum code
- Miscellaneous, GUI, and so on

Optimizing SIFT Tables

The overhead of the SIFT tables is massive and should be carefully considered for activation. Microsoft Dynamics NAV by default activates the SIFT tables when users create a new index with SumIndexFields. Users should review all their existing SIFT indexes and decide if they really need to keep them. They can de-activate the creation and maintenance of a SIFT table 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 supporting the retrieval of the cumulative sum, Microsoft Dynamics NAV will ask SQL Server to calculate the sum itself. For example, if they 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 "CRONUS International Ltd_$37$0" will be created and maintained. When a CALCSUM is used to display a FlowField in Microsoft Dynamics NAV showing the sum of all Sales Lines for a specific Sales Header (Order ORD-980001), the resulting query will look like this:

```
SELECT SUM(s29) FROM
  "CRONUS International Ltd_$37$0"
WHERE
  "bucket" = 2 AND
  "f1" = 1 AND
  "f3" = 'ORD-980001'
```

If they disable the SIFT table by clearing the MaintainSIFTIndex checkbox, Microsoft Dynamics NAV will still work, and the resulting query will look like this:

```
SELECT SUM("Amount") FROM
  "CRONUS International Ltd_$Sales Line"
WHERE
  "Document Type" = 1 AND
  "Document No_" = 'ORD-980001'
```

This is a very light on CPU overhead compared to the massive costs of maintaining the SIFT table.

However, SIFT tables are extremely beneficial when users need to sum up a larger number of records. With that in mind, users can check existing SIFT tables and see if they need some of the level of details. There is no need, for example, to store cumulative sum of just several records. Users can use the property SIFTLevels and disable specific levels by clearing the checkbox Maintain for a specific bucket, reducing the overall overhead of the SIFT table, but still keeping it in place for the larger number of records sum. On the other side of the spectrum, there is no need, for example, to keep cumulative sums on the top level buckets if they do not make business sense and would be never used, such as a total of Quantity on "Location Code" in the Item Ledger Entry table, since users would always filter on "Item No."

Optimizing Indexes

The second largest typical Microsoft Dynamics NAV overhead is the cost of indexes. The Microsoft Dynamics NAV database is over-indexed, since customers require certain reports to be ordered in different ways, and the only way to do it is to create a Microsoft Dynamics NAV key to sort data in these specific ways. However, SQL Server can sort results itself, and quite fast if the set is small, so there is no need to keep indexes for sorting purposes only. For example, in the Warehouse Activity Line table, there are a number of keys that begin with "Activity Type" and "No." fields, such as:

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

The issue here is that these indexes are not needed on SQL Server, because the Microsoft Dynamics NAV code always filters on "Activity Type" and "No." when using these keys. When it comes to SQL Server, the Query optimizer will look at the filter and realize that the clustered index is "Activity Type,No_,Line No_" and that the set is small, and that there is therefore no need to use an index to retrieve the set and return it in that specific order. It will use only the clustered index for these operations.

Additionally, the entire functionality is not used by customers, so if they never pick the stock by "Sorting Sequence No." for example, there is no need to maintain the index.

Developers should analyze the existing indexes with a focus on use and benefits compared to the overheads, and decide what action is need. Disable the index entirely using key property Enable, using the KeyGroups property, or using the MaintainSQLIndex property. Indexes that remain active can change structure using the SQLIndex property. Developers can also make the table clustered by a different index.

Enabled Property

Enabled property simply turns he specific key on and off t. It might have been there for temporary reasons and is no longer needed. If a key is not enabled and is referenced by a C/AL code or CALCSUMS function, users will get a run-time error.

KeyGroups Property

Use this property to select the (predefined) key groups to which the key belongs. Once users assign this key to one or more key groups, they can selectively activate or deactivate the keys of various groups by enabling and disabling the key groups.

To make use of the key groups for sorting, choose the Key Groups option on the Database Information window which appears when they select File, Database, Information, and then press the Tables button. There are key groups that are defined already, such as Acc(Dim), Item(MFG), and so on, but users can create more themselves and assign them to keys they want to control this way.

The purpose of key groups is to make it possible to set up a set of special keys that are used very seldom (such as for a special report that is run once every year). Since adding lots of keys to tables will eventually decrease performance, using key groups makes it possible to have the necessary keys defined, but only active when they are really going to be used.

MaintainSQLIndex Property

This property determines whether an SQL Server index corresponding 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 in order to sort 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 will be faster, but modifications to the table will be slower.

The more indexes there are on a table, the slower the modifications become. In situations where a key must be created to allow only occasional sorting (for example, when running infrequent reports), users can disable this property to prevent slow modifications to the table.

SQLIndex Property

This property allows users to define the fields that are used in the SQL index. The fields in the SQL index can:

- Be different than the fields defined in the key in Microsoft Dynamics NAV - there can be fewer fields or more fields.
- Be arranged in a different order.

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

When users define the SQL index for the primary key, it must include all the fields defined in the Microsoft Dynamics NAV primary key. Users can add extra fields and can still rearrange the fields to suit their needs.

Clustered Property

Use this property to determine which index is clustered. By default the index corresponding to Microsoft Dynamics NAV primary key will be made clustered.

Implicit/Explicit Locking

There are further considerations users should make when working with Microsoft Dynamics NAV on SQL Server. Microsoft Dynamics NAV has been designed to read without locks and locking only if needed, following optimistic concurrency recommendations. If users know that they going to modify records, they should indicate that intent to the driver (use explicit locking), so that the data is read properly to start with.

If they do not do so, there can be another bad behavior coming from this that if there was a cursor created for reading the data then it is impossible to change the nature of the cursor in the middle of using it, a new cursor has to be built for new isolation level (leading to performance problems with "NEXT," a situation that will be discussed in other sections).

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 timestamp = TS1)
TableX.Field1 := Value;	
TableX.MODIFY;	SELECT * FROM TableX WITH (UPDLOCK, REPEATABLEREAD) (the retrieved record timestamp = TS2) performs the update UPDATE TableX SET Field1 = Value WITH (REPEATABLEREAD) WHERE TimeStamp <= TS1

The reason for such a complex execution is that:

- The data was read with READUNCOMMITED isolation level, but because users are going to update, they need to make sure that they read committed data and issue an update lock on it to prevent other users from updating the same records.
- The data was originally read uncommitted; users need to lock the record and also make sure that they update the record with an original TimeStamp. If somebody else changes the record, that person will get an error: "Another user has modified the record since users retrieved from the database…."

Explicit Locking

However, if users actually indicate to the database driver that their intention is to modify the record, using explicit locking, they can eliminate the bad behavior entirely, as shown by the pseudo code below:

Sample code	Result
TableX.LOCKTABLE;	Indicates to the driver explicit lock
TableX.FIND('-');	SELECT * FROM TableX WITH (UPDLOCK) (the retrieved record timestamp = TS1)
TableX.Field1 := Value;	
TableX.MODIFY;	UPDATE TableX SET Field1 = Value WITH (REPEATABLEREAD) (the retrieved record timestamp is guaranteed to be TS1)

Problems With NEXT

In some situations, the NEXT command causes the biggest performance problem in Microsoft Dynamics NAV, and users should pay particular attention to avoid these situations. The problem is that the database driver is using a cursor, and there is no way to change the isolation level in the middle of the data retrieval. Also, if users change the condition of the set, then the set has to be retrieved again.

There are a number of scenarios that cause problems with NEXT. Instead, FETCH should be used, but if the set is wrong, a new SELECT statement needs to be executed. This imposes a serious performance penalty in SQL Server and in some situations leads to very lengthy execution.

Performance problems occurs if users perform a NEXT on a set, but they:

- Changed filter
- Changed sorting
- Changed key value
- Changed isolation level
- Performed NEXT "in the middle of nowhere" on a record that they got through GET for example.

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 "in the middle of nowhere"

Code	Result
SETRANGE(FieldA,Value);	
FIND('-');	Set based on filter on FieldA
REPEAT	
…	
SETRANGE(FieldB,FieldB);	New set based on filter on FieldA AND FieldB
…	
FIND('+');	Record position is end of the set
…	
SETRANGE(FieldB);	Filter removed
…	
UNTIL NEXT = 0	The system is asked go to NEXT from undefined position

Non-cursor to cursor

Code	Result
SETRANGE(FieldA,Value);	
FINDFIRST;	Non-cursor fetch
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 these solutions::

- Use a separate looping variable.
- Restore original key values, sorting, and filters before the NEXT statement.
- Read records to temporary tables, modify within, and write back afterwards.
- FINDSET(TRUE,TRUE).

Note also that FINDSET(TRUE,TRUE) is not a "real solution," it is merely a reduction of the costs and should be used only as a last resort.

Suboptimum Coding and Other Performance Penalties

Taking performance into consideration often influences programming decisions. For example, if users do not use explicit locking, or if they program bad loops and provoking problems with **NEXT**, they often pay a big price in terms of performance. They should also review their code and see how many times the code is reading the same table, or use **COUNT** for checking if there is a record **(IF COUNT = 0, IF COUNT = 1),** or using **MARKEDONLY** instead of pushing records to a temporary table and reading then from there. Additionally, there are features in the application that need to be avoided or minimized. The advance dimensions functionality is very cost demanding, for example. Users should review the application setup for the performance aspect and make corrective actions if they can.

C/SIDE setup can also make a big difference, such as object cache size, but a big overhead may come from the “Find As You Type” feature, for example, as well. Lastly, the GUI overhead can be slowing the client down, if, for example, a dialog is refreshed 1000 times in a loop. GUI overhead can also cause increased server trips – when users use the default **SourceTablePlacement = <Saved>** on forms, it costs more than using **Last** or **First**. Users should review all forms showing data from large tables to look for performance problems.

Overview of NDBCS

NDBCS, or the Microsoft Dynamics NAV Database Driver for Microsoft SQL Server, translates Microsoft Dynamics NAV data requests into Transaction-SQL (T-SQL). This translation allows the Microsoft Dynamics NAV clients to communicate with SQL Server.

For example, consider the following C/AL statements:

```
GLEntry.SETCURRENTKEY("G/L Account No.","Posting Date");
GLEntry.SETRANGE("G/L Account No.",'7140');
IF GLEntry.FIND('-') THEN;
```

This code causes the following information to be sent from Microsoft Dynamics NAV to the NDBCS driver:

Function Name	Parameter	Data
FIND/NEXT	Table	G/L Entry
FIND/NEXT	Search Method	-
FIND/NEXT	Key	G/L Account No.,Posting Date,Entry No.
FIND/NEXT	Filter	G/L Account No.:7140

The NDBCS driver often uses cursors. Cursors are used in SQL Server if you need to read records from a set record-by-record rather than retrieving an entire set. There are different types of cursors which have different capabilities. For example, the forward-only cursor allows fast reading from top to bottom, while the dynamic cursor allows retrieval in either direction. (There are two other types of cursors: static cursors and keyset-driven cursors.)

The NDBCS driver translates the preceding example into a set of T-SQL statements that uses a dynamic cursor, and looks like this:

```
SELECT * FROM "CRONUS International Ltd_$G_L Entry"
  WITH (READUNCOMMITTED)
  WHERE (("G_L Account No_"='7140'))
  ORDER BY "G_L Account No_","Posting Date","Entry No_"
  OPTION (FAST 5)
```

If you explicitly indicate that the records will be modified using the GLEntry.LOCKTABLE command followed by the C/AL code above, the following will be sent to the driver:

Function Name	Parameter	Data
LOCKTABLE	Table	G/L Entry
FIND/NEXT	Table	G/L Entry
FIND/NEXT	Search Method	-
FIND/NEXT	Key	G/L Account No.,Posting Date,Entry No.
FIND/NEXT	Filter	G/L Account No.:7140

In this case the driver will do the same, but use a different isolation level, like this:

```
WITH (UPDLOCK, ROWLOCK)
```

You can add a loop, like this:

```
IF GLEntry.FIND('-') THEN
  REPEAT
  UNTIL GLEntry.NEXT = 0;
```

As a result, the following will be sent to the driver when the first NEXT command is executed:

Function Name	Parameter	Data
FIND/NEXT	Table	G/L Entry
FIND/NEXT	Search Method	>
FIND/NEXT	Key	G/L Account No.='7140',Posting Date='01/01/00',Entry No.='77'

The second NEXT will produce this:

Function Name	Parameter	Data
FIND/NEXT	Table	G/L Entry
FIND/NEXT	Search Method	>
FIND/NEXT	Key	G/L Account No.='7140',Posting Date='01/02/00',Entry No.='253'

The important facts are that the NDBCS driver:

- Does not know what the code intends to do (whether you want to browse forward or backward through the table, or just want the first record).
- Builds dynamic cursors with many different optimizations, such as reading ahead using FETCH 5, FETCH 10, and so on, using read ahead through SELECT TOP 49 * FROM, dropping dynamic cursor and using forward-only cursor, and so on

Optimization of Cursors

By default, the way the dynamic cursors are used is not very efficient. Because cursors have a big impact on performance, handling them in a different way can yield significant improvements. For example, there is no reason to create a cursor at all for retrieving a single record. When they are necessary, however, they can still be efficient. Optimizing cursors can be done with these four Microsoft Dynamics NAV commands:

- ISEMPTY

- FINDFIRST
- FINDLAST
- FINDSET

The following topics discuss how each of these commands can be used.

ISEMPTY

By default, determining if a set is empty uses cursors. For example:

```
Customer.SETRANGE(Master, TRUE);
IF NOT Customer.FIND('-') THEN
  ERROR('Customer master record is not defined');
```

This code determines if a record exists, but causes the NDBCS driver to generate T-SQL that uses cursors. However, the ISEMPTY command has a different effect:

```
Customer.SETRANGE(Master, TRUE);
IF Customer.ISEMPTY THEN
  ERROR('Customer master record is not defined');
```

When executed, the code above results in this T-SQL command:

```
SELECT TOP 1 NULL FROM …
```

Note that the NDBCS driver uses NULL, which means that no record columns are retrieved from the database (as opposed to '*', which would get all columns). This makes it an extremely efficient command that causes just a few bytes to be sent over the network. This can be a significant improvement as long as subsequent code does not use the values from the found record.

FINDFIRST

Retrieving the first record in a table can also be an unnecessarily expensive command. Consider this code:

```
Customer.SETRANGE(Master, TRUE);
IF NOT Customer.FIND('-') THEN
  ERROR('Customer master record is not defined');
```

The FINDFIRST command retrieves the first record in a set. Like ISEMPTY, FINDFIRST does not use cursors:

```
Customer.SETRANGE(Master, TRUE);
IF NOT Customer.FINDFIRST THEN
  ERROR('Customer master record is not defined');
```

When executed, this T-SQL is generated:

```
SELECT TOP 1 * FROM ... ORDER BY ...
```

Warning: If you want to do a REPEAT/UNTIL NEXT loop, you should not use this command, as the NEXT will have to create a cursor for fetching the subsequent records.

FINDLAST

Retrieving the last record in a table can also be an unnecessarily expensive command. Consider this code:

```
Message.SETCURRENTKEY(Date);
IF Message.FIND('+') THEN
  MESSAGE('Last message is dated ' + FORMAT(Message.Date));
```

The FINDLAST command retrieves the last record in a set. Like FINDFIRST, FINDLAST does not use cursors:

```
Message.SETCURRENTKEY(Date);
IF Message.FINDLAST THEN
  MESSAGE('Last message is dated ' + FORMAT(Message.Date));
```

This command retrieves the last record in the set, and does not use cursors. When executed, this T-SQL is generated:

```
SELECT TOP 1 * FROM ... ORDER BY ... DESC
```

Warning: If you want to do a REPEAT/UNTIL NEXT(-1) loop, you should not use this command, as the NEXT will have to create a cursor for fetching the subsequent records.

FINDSET

The FINDSET allows browsing through a set of records. In previous versions, retrieving a set in a loop could only be done this way:

```
IF FIND('-') THEN
  REPEAT
  UNTIL NEXT = 0;
```

FINDSET can be used in the following manner without any arguments (FINDSET arguments are explained later in this chapter), like this:

```
IF FINDSET THEN
  REPEAT
  UNTIL NEXT = 0;
```

Unlike the FIND('-') command, FINDSET does not use cursors. When executed, the T-SQL result looks like this:

```
SELECT TOP 500 * FROM ...
```

The REPEAT/UNTIL NEXT will browse through the records locally on the client machine. This is the recommended way to retrieve sets quickly -- without any cursor overhead.

Maximum Record Set Size

There is a parameter in Microsoft Dynamics NAV where you can set up the maximum of how many records will be retrieved from the database (File, Database, Alter, Advanced tab, Caching, Record Set = 500). If your set is bigger than the maximum, Microsoft Dynamics NAV will carry on working but replace the reading mechanism with a dynamic cursor. Therefore, if you know that it will happen, you should use the 'old' FIND('-') command as opposed to FINDSET. Secondly, you can only use FINDSET for forward direction; it will not work for REPEAT/UNTIL NEXT(-1) . Thirdly, if you use the LOCKTABLE command prior to the FINDSET, the set will be locked, and records can be modified within the loop.

A good example of an efficient use of cursors (using the 'old' FIND command), is when you read a big set of records, for example all G/L Entries for a specific account, probably with more that 500 records in the set:

```
GLEntry.SETRANGE("G/L Account No.", "6100");
IF GLEntry.FIND('-') THEN
  REPEAT
  UNTIL GLEntry.NEXT = 0;
```

A good example of using the new FINDSET command (as opposed to using the 'old' FIND command), is when you read a small set of records, such as all sales lines in a sales order, probably always with less then 500 records. This can be done this way:

```
SalesLine.SETRANGE("Document Type","Document Type"::Order);
SalesLine.SETRANGE("Document No.",'S-ORD-06789');
IF SalesLine.FINDSET THEN
  REPEAT
    TotalAmount := TotalAmount + SalesLine.Amount;
  UNTIL SalesLine.NEXT = 0;
```

FINDSET(TRUE)

This variation of FINDSET locks the set that is read, so it is equivalent to a LOCKTABLE followed by FIND('-). FINDSET(TRUE) can be used like this:

```
IF FINDSET(TRUE) THEN
  REPEAT
  UNTIL NEXT = 0;
```

Unlike the commands discussed previously, this command does use a dynamic cursor. The main purpose of this command is to raise isolation level before you start reading the set because the resulting records are to be modified.

FINDSET(TRUE) uses the read-ahead mechanism to retrieve several records instead of just one. It is recommended that **LOCKTABLE** be used with FINDSET for small sets, and that the **FINDSET(TRUE)** command is used for sets larger than 500 records (the Record Set parameter).

A good example of using the **FINDSET(TRUE)** command is when you read a big set of records for modification. For example, when going through all G/L entries for a specific account, and changing a field value based on the record condition, the filtered set will probably have more than 500 records. This might be done this way:

```
GLEntry.SETRANGE("G/L Account No.", "6100");
IF GLEntry.FINDSET(TRUE) THEN
  REPEAT
    IF (GLEntry.Amount > 0) THEN BEGIN
      GLEntry."Debit Amount" := GLEntry.Amount;
      GLEntry."Credit Amount" := 0
    END ELSE BEGIN
      GLEntry."Debit Amount" := 0;
      GLEntry."Credit Amount" := -GLEntry.Amount;
    END;
    GLEntry.MODIFY
  UNTIL GLEntry.NEXT = 0;
```

A good example of using the **LOCKTABLE** and **FINDSET** command (as opposed to using the FINDSET(TRUE) command) is when you read a small set of records and need to modify those. For example, when going through all sales lines for a specific order, and changing several fields values, the filtered set will probably have less than 500 records. This can be done this way:

```
SalesLine.SETRANGE("Document Type","Document Type"::Order);
SalesLine.SETRANGE("Document No.",'S-ORD-06789');
SalesLine.LOCKTABLE;
IF SalesLine.FINDSET THEN
  REPEAT
    SalesLine."Qty. to Invoice" :=
      SalesLine."Outstanding Quantity;
    SalesLine."Qty. to Ship" :=
      SalesLine."Outstanding Quantity;
    SalesLine.MODIFY
  UNTIL SalesLine.NEXT = 0;
```

FINDSET(TRUE, TRUE)

This variation of the **FINDSET(TRUE)** allows the modification of the Key value of the sorting order of the set. It can be used like this:

```
IF FINDSET(TRUE,TRUE) THEN
  REPEAT
  UNTIL NEXT = 0;
```

The command, like **FINDSET(TRUE)**, uses a dynamic cursor. The main purpose of this command is to raise isolation level before you start reading the set because you want to modify the set. It will not use the read-ahead mechanism. Instead it retrieves one record at a time, because the set is expected to be invalidated within the loop. Avoid using this command, since the loop code should be changed to a more efficient method of working, such as using a different variable for browsing through the set.

A good example of using the **FINDSET(TRUE,TRUE)** command (as opposed to using **FIND** command) is when you read a set of records and need to modify a key value. This should be avoided by any means. If you do not have a way to avoid this, use **FINDSET(TRUE,TRUE)**. For example, going through all sales lines for a specific order, and changing key value, the filtered set will probably have less than 500 records in the set. This can be done this way:

```
SalesLine.SETRANGE("Document Type","Document Type"::Order);
SalesLine.SETRANGE("Document No.",'S-ORD-06789');
SalesLine.SETFILTER("Location Code",'');
IF SalesLine.FINDSET(TRUE,TRUE) THEN
  REPEAT
    IF SalesLine.Type = SalesLine.Type::Item THEN
      SalesLine."Location Code" := 'GREEN';
    IF SalesLine.Type = SalesLine.Type::Resource THEN
      SalesLine."Location Code" := 'BLUE';
    SalesLine.MODIFY
  UNTIL SalesLine.NEXT = 0;
```

Note that the example above can be easily changed into more efficient code, using **FINDSET** as opposed to **FINDSET(TRUE,TRUE)**, using a separate variable to modify the records. This can be done this way:

```
SalesLine.SETRANGE("Document Type","Document Type"::Order);
SalesLine.SETRANGE("Document No.",'S-ORD-06789');
SalesLine.SETFILTER("Location Code",'');
SalesLine.LOCKTABLE;
IF SalesLine.FINDSET THEN
  REPEAT
    SalesLine2 := SalesLine;
    IF SalesLine.Type = SalesLine.Type::Item THEN
      SalesLine2."Location Code" := 'GREEN';
    IF SalesLine.Type = SalesLine.Type::Resource THEN
      SalesLine2."Location Code" := 'BLUE';
    SalesLine2.MODIFY
  UNTIL SalesLine.NEXT = 0;
```

Locking, Blocking, and Deadlocks

Reading Uncommitted Data

When data is read from the database, Microsoft Dynamics NAV will use the READUNCOMMITTED isolation level, meaning that any other user can modify the records that are currently being read. This is often referred to as optimistic concurrency. Data that is read is considered “dirty” because it can be modified by another user. When the data is updated, the Microsoft Dynamics NAV driver must compare the timestamp of the record. If your record is ‘old’, the Microsoft Dynamics NAV error “Another user has modified the record after you retrieved it from the database” is displayed.

Optimistic concurrency allows for better performance because data can be accessed simultaneously by multiple queries. The tradeoff is that care must be taken when writing code that modifies the data. This requires that locking and blocking be employed to synchronize access, but deadlock – a condition where one or more competing processes are stalled indefinitely – can occur. The following subtopics discuss strategies for synchronizing data access while avoiding deadlock.

Locking

The isolation level can be changed to a more restrictive setting, such as **UPDLOCK**. In this level, records that are read are locked, meaning that no other user can modify the record. This is referred to as *pessimistic locking*, and causes the server to protect the record in case you want to modify it -- making it impossible for others to modify.

An example of a lock of a customer record can be demonstrated by this code:

```
Customer.LOCKTABLE;
Customer.GET('10000');          ← Customer 10000 is locked
Customer.Blocked := TRUE;
Customer.MODIFY;
COMMIT;                         ← Lock is removed
```

If you do not lock the record, 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 := TRUE; Customer.MODIFY; COMMIT;	User B modifies record
Customer.Blocked := FALSE; 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 will be blocked and have to wait. If they wait longer than the defined timeout, they will get a Microsoft Dynamics NAV error: "The XYZ table cannot be locked or changed because it is already locked by the user with User ID ABC." You can change the default timeout with File, Database, Alter, Advanced tab, Lock Timeout checkbox and Timeout duration (sec) value.

Based on the previous example where two users try to modify the same record, you can lock the data you intend to modify, preventing other users from doing the same. Here 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 := TRUE; Customer.MODIFY;	… blocked, waiting …	User A successfully modifies record.
COMMIT;		Lock is released.
	…	Data is sent to user B
	Customer.Blocked := FALSE; Customer.MODIFY;	User B successfully modifies record.
	COMMIT;	Lock is released.
SUCCESS	SUCCESS	

Deadlock

There is a potential situation when blocking cannot be resolved by the server in a nice way. The situation arises when one process is blocked because another process has locked some data. The other process is also blocked because it tries to lock the first process data. Only one of the transactions can be finished; SQL Server will terminate the other and send an error message back to the client: "Your activity was deadlocked with another user …"

For example, consider a case where two users are working simultaneously and trying to get each other's blocked records, as shown in this 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, while B wants A's record → Conflict
"Your activity was deadlocked with another user"		SQL Server detects deadlock and arbitrarily chooses one over the other, so one will receive an error.
ERROR	SUCCESS	

SQL Server supports record level locking, so you might have a situation where these two activities will bypass each other without any problems, such as with this pseudo code (note that User A is fetching the last record compared to the situation above):

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	

Note that there would be a deadlock if one of the tables was empty, or contained one record only. To add to this complexity, you may have a situation where two processes read the same table from opposite directions and meet in the middle, such as with this 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, while B wants A's record → Conflict
	"Your activity was deadlocked with another user"	SQL Server detects deadlock and chooses 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 hard to avoid. The good news is that the transaction chosen to fail will be rolled back to the beginning, so apart from the user not being happy, there should be no major issue. However, if the process has been 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 your customer.

Avoiding Deadlocks

A large number of deadlocks can lead to major customer dissatisfaction, but deadlocks cannot be avoided entirely. To reduce the number of deadlocks, you should make sure that you:

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

If the above is not possible due to the complexity of the processes, as a last resort, you can revert to serializing the code by making sure that conflicting processes cannot execute in parallel. The following code demonstrates how this can be done:

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, a higher probability of timeouts can be experienced, so keeping the length of transactions short becomes even more important. This also demonstrates that you can combine the various principles and methods all together, depending on the situation and complexity; one method works for one customer while the other works for other.
You should also adhere to some of the following "golden rules":

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

Finally, if it is too complex or you have limited time, consider discussing with the customer the possibility of overnight processing of heavy jobs, thus avoiding the daily concurrency complexity and avoiding the high costs of rewriting the code.

How SIFT Data is Stored in SQL Server

SumIndexField is always associated with a key, and each key can have a maximum of 20 SumIndexFields associated with it. When you set the MaintainSIFTIndex property of a key to Yes, Microsoft Dynamics NAV will regard this key as a SIFT key and create the SIFT structures that are needed to support it.

Any field of the Decimal data type can be associated 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 normal table, but is called a SIFT table. As soon as the first SIFT table is created for a base table, a dedicated SQL Server trigger is also created and is then automatically maintained by Microsoft Dynamics NAV. This is known as a SIFT trigger. A base table is also a standard Microsoft Dynamics NAV table, as opposed to an extra 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 are created to support this base table. The purpose of the SIFT trigger is to implement all the modifications that are made on the base table whenever a SIFT table is affected. This means that the SIFT trigger automatically updates the information in all the 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".

A SIFT table is created for every base table key that has at least one SumIndexField associated with it. No matter how many SumIndexFields are associated with a key, only one SIFT table is created 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 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 along with the SumIndexFields that are associated with this SIFT key. But the first column in every SIFT table is always named "bucket" and contains the value of the bucket or SIFT level for the precalculated sums that are stored in the table. You can see the structure if you look at the SIFTLevels property for a key in Microsoft Dynamics NAV.

After the bucket column is a set of columns with names that start 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 following 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 3 in the base table G/L Entry).

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

The name of these columns has the following 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.

With regards to performance, SIFT tables are one of the biggest Microsoft Dynamics NAV performance problems on SQL Server, as one record update in the base table produces a potentially massive stream of I/O requests with updates to the records in the SIFT tables, possibly blocking other users during that time.

Conclusion

This chapter covered the essential points of insuring optimal performance in Microsoft Dynamics NAV applications, particularly when using SQL Server 2005. The two database options were compared, revealing that they are different database types and therefore have different performance characteristics.

The specific ways in which SQL Server is implemented were discussed in detail to reveal why some operations are expensive in terms of performance, and others are cheap and just as effective.

Quick Interaction: Lessons Learned

Take a moment to write down three Key Points you have learned from this chapter:

1.

2.

3.