Tampere University

Ameer Kulabi

INCREASING DATABASE PERFORMANCE WITH MEMORY-OPTIMIZED TABLES

Case M-Files

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
November 2024

Abstract

Ameer Kulabi: Increasing database performance with memory-optimized tables: Case M-Files
Master's thesis
Tampere University
Master's Degree Programme in Software Development
November 2024

Databases are a critical part of many software applications, and database performance affects the overall performance of those applications. In the 1980s, when the first relational database system was developed, the concept of cheap disk and expensive main memory led to the decision to store data on disk. Nowadays, that concept is no longer valid. However, retrieving data from the disk remains a slow operation. In 2014, Microsoft published its in-memory OLTP engine, which was fully integrated into SQL Server. The in-memory OLTP engine supports memory-optimized tables, which store data in main memory.

M-Files, the thesis case company, is a software company specializing in enterprise information management. It uses its application to bring value to its customers by digitizing their processes. During this thesis, the tables of the M-Files vault database are migrated to be memory-optimized, and the performance of the old and new systems are compared. The implementation is not meant for customers, as the goal of the study is to examine the potential performance benefits and identify blockers to migration.

During the migration process of the tables, many blockers that could hinder the use of memory-optimized tables are found. In the M-Files vault database, triggers and change-tracking features were used, neither of which are supported with memory-optimized tables. Additionally, some tables used data types that were incompatible with this new table type. That is why we need to redesign some tables to get the most out of this feature. In contrast, the performance comparison shows improvements in most measured operations.

Keywords: Memory-optimized tables, SQL Server, In-memory OLTP, Performance comparison

The originality of this thesis has been checked using the Turnitin Originality Check service.

Tiivistelmä

Ameer Kulabi: Increasing database performance with memory-optimized tables
Pro gradu -tutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan tutkinto-ohjelma
November 2024

Tietokanta on olennainen osa useimpia ohjelmistosovelluksia, ja tietokantojen suorituskyky vaikuttaa merkittävästi sovellusten kokonaissuorituskykyyn. 1980-luvulla, kun ensimmäiset relaatiotietokannat kehitettiin, halpa levymuisti ja kallis keskusmuisti johtivat siihen, että tietokantojen data tallennettiin levylle. Nykyään, keskusmuisti on halventunut, mutta tietojen hakeminen levyltä on edelleen hidas operaatio. Vuonna 2014 Microsoft julkaisi In-memory OLTP-tietokantajärjestelmän ydimen, joka on täysin integroitu osa SQL Serveriä. Tämä teknologia tukee muistioptimoitujen taulujen käyttöä, jotka pitävät niiden data koko ajan keskusmuistissa.

M-Files on ohjelmistoyritys, joka on erikoistunut tiedonhallintaan. Tässä tutkielmassa migroidaan M-Files-varastotietokannan tauluja muisti-optimoiduiksi tauluiksi ja vertaillaan lopuksi vanhaa ja uutta järjestelmää. Migroinnin aikana tutkitaan myös esteitä, jotka voivat vaikeuttaa tämän ominaisuuden käyttöä.

Migroinnin aikana, huomataan, että on olemassa monta ominaisuutta, joita M-Files varastotietokanta käyttää, mutta toisaalta muisti-optimoidut taulut eivät tue niitä. Käytetyt, mutta tuettomat ominaisuudet ovat muun muassa herättimet ja muutosten seuranta. Lisäksi muutamissa tauluissa on käytetty tietotyyppejä, joita muisti-optimoidut taulut eivät tue. Näiden perusteella voimme päätellä, että ominaisuuden käyttöönotto vaatii muutamien taulujen uudelleensuunnittelun. Huolimatta haasteista, tulokset osoittavat, että suurin osa tutkituista toiminnallisuuksista toimii nopeammin muisti-optimoiduilla tauluilla.

Avainsanat: Muisti-optimoidut taulut, SQL Server, In-memory OLTP, Suorituskykyvertailu

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck-ohjelmalla.

Preface

To begin, I would like to acknowledge the individuals who made this thesis possible.

At M-Files, I would like to thank Mika Hirvonen and Timo Partanen. Mika Hirvonen played a main role in managing official arrangements and proofreading the text. Timo Partanen provided this interesting topic for me to study and also took on the role of technical advisor throughout the research. Both provided essential guidance to ensure the study proceeded in the appropriate direction with the required resources.

At Tampere University, I am grateful to Marko Junkkari and Toni Taipalus for their invaluable feedback on the thesis. Moreover, their ideas were valuable to achieve a well-structured thesis.

Tampere, Finland

20.11.2024

Ameer Kulabi

Contents

1 Introduction 1

2 Relational Database Management Systems 4

2.1 Firebird 4

2.2 Microsoft SQL Server 4

2.2.1 Isolation Levels 7

2.3 Tools 9

3 Memory-Optimized tables 11

3.1 In-Memory OLTP Architecture 12

3.2 Row structure and Multi Versioning Technique 13

3.3 Indexes in Memory-Optimized tables 14

3.3.1 Hash indexes 15

3.3.2 Non-clustered indexes 16

3.3.3 Column-store indexes 16

3.4 Limitations 17

3.5 Off-Row Data 19

3.6 Table Altering 20

3.7 Hardware requirements 20

3.8 Customer Cases 21

4 M-Files Vault Database 23

4.1 M-Files Vault Operations 24

4.2 Transaction Performance Analysis Overview 25

4.3 Implementing changes 27

5 Method of research 30

6 Results and performance comparison 33

7 Discussion 37

8 Conclusion 41

9 References 43

A Appendix: Bench-marking scripts 47

LIST OF ABBREVIATIONS

SQL Structured querying language

OLTP Online transaction processing

CPU Central processing unit

SBCS Single-byte character set

MBCS Multibyte character set

DDL Data definition language

1 Introduction

Back in the 1980s, when the first relational database management systems were developed, the main memory was limited on computers that were running the databases [12]. Limited amounts of the main memory forced the storage of database data on the disk, with only part of the database loaded into the main memory when processed [12]. That was an acceptable decision at that time, because databases did not have the same performance requirements as some have now. However, nowadays loading data pages from the disk to the main memory adds an unnecessary workload to pretty active databases, which need every resource possible to achieve better response time.

The size of the main memory available on computers started to grow and in contrast, the price of the main memory decreased 10 times every 5 years [13]. This encouraged vendors to develop in-memory databases. The idea of in-memory databases is to keep all the database data in memory, thus eliminating the need to load data from the disk. Examples for in-memory database management systems are SAP HANA, Microsoft SQL Server Hekaton, and Oracle TenTimes. [6]

SAP HANA is column-store database system that stores its data in the main memory while having a copy of the data in more persistent storage to support high durable tables [15][28]. Because of its column-store nature it is more suitable for analytics queries that handles many rows and a few columns [15][28]. Microsoft SQL Server Hekaton is an in-memory OLTP engine that supports in-memory tables called memory-optimized tables [14]. Hekaton is completely integrated into SQL Server, and thus database developers can combine memory-optimized tables and disk-based tables under one database [14]. According to Oracle's official documentation, the fastest in-memory relational database is Oracle's TimesTen [24]. TimesTen In-memory database supports ACID properties and SQL like other relation database management systems [9][24]. However, it is a standalone in-memory database and developers cannot combine in-memory tables with disk-based tables like in Hekaton [9][24]. This thesis focuses on Microsoft's Hekaton.

Like other companies, M-Files, the case company of this study, is eager to improve the performance of its application for a better user experience and to handle large amounts of data in a reasonable time. One way to achieve this goal is to have a

faster database. This desire initiated the need for this thesis.

M-Files is an information management company that utilizes its software application, which has been developed over many years by multiple developers, to bring value to its customers by digitizing their processes. [22]

The case company decided to study one Microsoft SQL Server feature called memory-optimized tables. This study focuses on the performance benefits of utilizing this feature. To make the comparison possible, we must first migrate the existing M-Files vault database tables to be memory-optimized. Along the way, the main blockers to the migration are identified. In addition to the database changes, the SQL queries within the M-Files software should be adjusted accordingly to be compatible with the migrated database. At the end of this study, the performance of the existing software is compared with the new software that uses the newly migrated database. It was decided that we would compare four operations: document creation, document check-in, document modification, and searches. These were chosen because they are the primary and most used operations in M-Files software.

The literature overview part of this study is based on the SQL Server official documentation, books, articles, and M-Files user guide. Popular libraries like ACM, ProQuest, and Oreilly were used to search for suitable sources. Sources were filtered based on the publication year because the focus area of this study is under active development, and information quickly becomes outdated. Moreover, the availability of sources was limited.

The results show potential performance gains in several operations but not in all. With memory-optimized tables, object check-in, metadata modification, and object creation operations are faster. However, range searches are slower. Additionally, during the migration process of the database and the modification to the software, we noticed that the work required to implement such a feature is significant due to the current limitations of the feature. Limitations include the lack of change tracking, table triggers, and support for some data types. In addition to these difficulties, the indexes must be redesigned to achieve the best results. For more information about how indexes work with memory-optimized tables, check Subsection 3.3.

In the next section, Firebird and SQL Server are introduced, both of which are used by M-Files software. In the section after that, what memory-optimized tables are and the main differences between them and disk-based tables are explored. The limitations of the feature will be discussed. Section 4 explains the M-Files opera-

tions in more detail, including the changes required to make the M-Files software compatible with the new table type. Section 5 introduces the study method and the main problem. Section 6 presents the results, and the section after that will contain the observations made on the results and the migration process. In the last section, there is a summery, where the highlights of thesis are listed.

2 Relational Database Management Systems

M-Files has a complex and massive database with more than 300 tables. As a database management system, M-Files uses Microsoft SQL Server and Firebird. Firebird is used for on-premise and relatively small M-Files vaults, while Microsoft SQL Server is used for cloud-based and larger ones. The database's data model does not follow known best practices for relational database models. For example, the same data is stored in many tables. The data model has been designed this way on purpose to achieve faster searches. Next, Firebird is briefly explored and then in more details Microsoft SQL Server is explained, as it is the primary focus of this study.

2.1 Firebird

Firebird is an open-source relational database management system. Initially, it was developed by Inprise Corp, now known as Borland Software Corp, using C and C++. Its main features include support for various operating systems, the ability to handle large volumes of data, and support for stored procedures and triggers. Additionally, there are several tools that database administrators utilize to manage the databases. Other features include a small footprint, high performance, easy integration, security, maintenance, and support. [5]

2.2 Microsoft SQL Server

SQL Server is a relational database management system (RDBMS) designed to store, manage, and retrieve data efficiently. The development of SQL Server began in 1988 by Microsoft, Ashton-Tate, and Sybase. The first version was published in 1989. Microsoft took complete control of development in 1994, after which it began rewriting the codebase, culminating in 1998. It supports SQL-like querying language and ACID transactions like any RDBMS. Due to its general approach, SQL Server can be used as a database for various industry applications, from healthcare and finance to e-commerce and business intelligence. [18]

SQL Server stores and manages structured data in tables. Tables are groups of rows that have the same structure and represent data related to objects. Tables

are logically linked through relationships, ensuring integrity and facilitating efficient retrieval. [18]

The querying language for SQL Server is called Transact-SQL (T-SQL) [21][25]. Users can construct queries to retrieve specific data subsets, filter results, perform calculations, and manipulate data within the database [21][25]. It follows typical SQL syntax [21][25]. As with any other programming, T-SQL is divided into several basic objects: literal values, identifiers, delimiters, comments, and reserved keywords [25]. Literal values are unchanging values within the SQL statements [21][25]. Literal values can be numeric, string, date, time, boolean, or Binary literals [21][25]. Identifiers are used as names for database objects [21][25]. Database objects are the main components or structures from which the database is constructed [21][25]. These objects can be divided into two main categories Data Definition Language (DDL) objects and Data Manipulation Language (DML) Objects [21][25]. Tables, views, indexes, and stored procedures are examples of DDL objects, while cursors and replication objects are examples of DML objects [21][25]. When identifiers are surrounded by double quotation marks, they are called delimited identifiers [21][25]. Those delimiters allow the use of reserved keywords and space in the names of database objects [21][25]. As with any other programming and querying language, T-SQL also supports comments. Everything between /* */ is considered a comment and can be used for multiline comments [21][25]. However, two hyphens (–) mark everything after them in the same line as a comment [21][25]. Reserved keywords are words that have a specific meaning defined when the language was developed and cannot be used to refer to something else [21][25]. For example, these cannot be used as a table name unless marked with delimiters [21][25]. In table 2.1, the full list of the supported data types is found.

In addition, there are other less commonly used data types like cursor, image, geography, geometry, hierarchyid, json, rowversion, table, uniqueidentifier. It is worth mentioning that the number of bytes for string data types does not necessarily match the number of characters stored. The number of characters stored depends on whether SBCS (single-byte character set) or MBCS (multibyte character set) is used. [21]

SQL Server has supported the In-Memory OLTP feature since 2014 [4][21][7]. It involves storing frequently accessed data in memory, thus eliminating the overhead of reloading data from disk to main memory [4][7][21]. Its structure also eliminates the need for latches and locks [4][7][21]. This feature is the focus of this study and will be further examined in Section 3. In the next subsection, isolation levels are

introduced, with a particular focus on in-memory tables.

Table 2.1 *Information listed are present in the official documentation of Microsoft SQL Server [21].*

tinyint	1 byte	Suitable for such use cases where the number is between 0 and 255.
smallint	2 bytes	Supports larger range then tinyint from -32768 to 32767.
int	4 bytes	Uses 4 bytes and thus supports from -2147483648 to 2147483647.
bigint	8 bytes	Uses 8 bytes and has the largest range among other exact-number data types from -9223372036854775808 to 9223372036854775807.
bit	1 bit	Multiple bit typed columns can be combined and stored as 1 or more bytes. For example, if there are 8 or fewer bit columns in a table, the columns will be stored as 1 byte. This type is also used to store Boolean values. True is converted to 1 and False converted to 0.
decimal and numeric	5 - 17 bytes	The precision of the number affects on the total memory consumed. The larger precision is the bigger the size is. Precision is the maximum of decimal digits to be stored.
money and smallmoney	4-8 bytes	Both are used to store monetary or currency values. Money allocates 8 bytes and its range is -922337203685477.5808 to 922337203685477.5807, while smallmoney needs 4 bytes and its range is -214748.3648 to 214748.3647.
float and real	4 - 8 bytes	Approximate-number data types and the size of the data depends on the precision where 7 digits precision is called real and 15 digits precision is called double precision.
date	3 bytes	Was introduced first in SQL Server 2008 and it supports many string literal formats such as [m]m/dd/[yy]yy, [m]m-dd-[yy]yy, and [m]m.dd.[yy]yy.
time	5 bytes	Used to store a time of a day. It supports many string literal formats too.
datetime and datetime2	6 - 8 bytes	Define a date combined with a time of a day. Datetime2 supports larger date range than datetime.

datetimeoffset	10 bytes	In addition to time and date, datetime-offset also stores information about the timezone.
smalldatetime	4 bytes	This type also stores date and time, however seconds and fractional seconds are always zero and it does not have fractional seconds.
char	1 - 8000 bytes	It is used for fixed-size string. The maximum number of characters that can be stored in this data type depends on the encoding character set.
varchar	1 - $(2^{31} - 1)$ bytes	Unlike char, this data type is used to store variable-size string. Maximum size that can be stored in this type is 2 GB.
text	1 - $(2^{31} - 1)$ bytes	Also, stores vairable-length non-unicode string. However, this is data type will be removed in the future version of SQL Server along with ntext and image.
nchar	1 - 8000 bytes	Similar to char, but it stores Unicode data.
nvarchar	1 - $(2 * 2^{31} - 1)$ bytes	Similar to char, but it stores Unicode data.
ntext	1 - $(2^{30} - 1)$ bytes	Similar to text, but it stores Unicode data.
binary	1 - 8000 bytes	It is for fixed-length binary data.
varbinary	1 - $(2^{31} - 1)$ bytes	It also stores binary data, however unlike binary data type it is variable length.
xml	1 - $(2^{31} - 1)$ bytes	It is used to store XML data.
sql_variant	1 - 8000 bytes	It supports storing values of various SQL Server data types.

2.2.1 Isolation Levels

Transactions, or in other words, units of work in Microsoft SQL Server, follow the ACID principles. ACID stands for atomicity, consistency, isolation, and durability. Atomic transactions either commit or roll back all the changes. Consistency ensures that systems and changes do not violate any rules or constraints. Isolation ensures that transactions see only committed changes and do not access changes from uncommitted transactions. Durability ensures that all committed changes are stored permanently. All these characteristics are supported in both disk-based and

Isolation Levels for Disk-Based Tables	Isolation Levels for Memory-Optimized Tables
READ UNCOMMITTED, READ COMMITTED, READ COMMITTED SNAPSHOT	SNAPSHOT, REPEATABLE READ, SERIALIZABLE
REPEATABLE READ, SERIALIZABLE	SNAPSHOT only
SNAPSHOT	Not supported

Figure 2.1 *Allowed isolation level combinations in cross-container transactions [8]*

in-memory tables [23]. [8][18]

Isolating different transactions in multi-user environments, particularly in disk-based tables, is a complex task. The main challenge lies in balancing performance and data consistency. Locking is often used to meet isolation requirements, but this can lead to performance issues in highly active systems. To address this, different isolation levels have been introduced. It is crucial for developers and administrators to carefully study the use case to determine the most suitable isolation level. [8]

Concurrency issues, which are effectively managed by isolation levels, can lead to significant problems if not properly addressed. These issues include dirty reads, non-repeatable reads, and phantom reads. Dirty reads occur when one transaction reads uncommitted changes made by another transaction, potentially leading to inconsistent data. Non-repeatable reads happen when the same query in a single transaction returns different results, causing confusion and potential errors. The phantom phenomenon occurs when other transactions insert new values in the range that the current transaction is retrieving, causing the same query to return more rows after the change than before. Understanding these issues and the isolation levels that address them is crucial for maintaining data integrity and system reliability. [2][8]

The different isolation levels in disk-based tables are read uncommitted, read committed, repeatable read, serializable, and snapshot. All the concurrency issues mentioned are possible at the read uncommitted isolation level, as no locks are added. Read committed prevents only dirty reads. Repeatable read prevents both dirty reads and non-repeatable reads. Serializable and snapshot levels prevent all the presented issues. All the mentioned levels use locks to prevent these issues, except for the snapshot level, where transactions only see changes committed before they execute. [2][8]

There are three isolation levels in memory-optimized tables: Snapshot, repeatable read, and serializable. Since memory-optimized tables never use locks, the implementation of those isolation levels differs from that of disk-based tables. Although the snapshot isolation level is implemented differently, it functions exactly the same as it does in disk-based tables. [8]

In cross-container transactions, not all isolation level combinations are possible [8]. As previously mentioned in this study, disk-based and in-memory tables can be accessed through the Query Interop Engine [8]. This allows developers to write transactions that execute queries on both table types [8]. These transactions are called cross-container transactions [8]. Each table can have its own isolation level [8]. However, not all combinations are possible [8]. See Figure 2.1 for more information on which combinations are possible.

2.3 Tools

For companies, identifying potential database tables that may benefit from the migration of memory-optimized tables is an exploratory process. To estimate the required work, companies use tools such as the Transaction Performance Analysis report in SMSS (Microsoft SQL Server Management Studio), Memory Optimization Advisor, and Native Compilation Advisor [21][29].

The transaction Performance Analysis report can be generated via Microsoft SQL Server Management Studio. It provides information about which parts of the database could benefit from the features of in-memory tables [29]. For tables, the report is divided into three parts. The first part consists of scan statistics, such as the percentage of total access and Lookup Statistics/Range Scan Statistics. The first value indicates how much a specific table has been accessed. A high value suggests that the table may see performance improvements. The second value indicates how many lookups and range scans are performed on the target database table. The second part of the report includes information about the latches and locks. It contains values such as Percent of total waits, Latch Statistics, and Lock Statistics. The percentage of total waits is based on the latch and lock waits on the target database table compared with the overall database activity. Latch Statistics clarifies the number of latches that wait for executed queries using this table, while Lock Statistics records the number of lock acquisitions and waits of the queries related to the target table. The last part of the report addresses Migration Difficulties that could be encountered during the conversion phase. [21]

In addition to the Transaction Performance Analysis report, Microsoft SQL Server Management Studio provides another beneficial feature: the In-Memory OLTP Migration Checklist, which lists the features of the target database table that are not supported by memory-optimized tables. [29]

3 Memory-Optimized tables

This section provides background information to help the reader understand the term memory-optimized tables. Memory-optimized tables were introduced in response to the increasing performance requirements for querying data from extremely busy databases expected to handle substantial amounts of data within a reasonable time. Memory-optimized tables are also referred to as in-memory tables in some sources [1][14]. Both terms will be used interchangeably in this study.

As data growth became a problem, developers began speeding up CPU clocks and adding more CPUs:s and cores. However, this approach can only be applied for a limited time before reaching its limit. As a result, developers started optimizing their code to generate faster queries. The situation worsened because disk-based tables use latches and locks to protect data from concurrent access, leading to additional CPU instructions that slow down the system. To address this problem, Microsoft decided to build memory-optimized tables using a different architecture. Microsoft began developing this new feature in 2011, and it was released in 2014. When they started implementing this feature, they aimed to achieve a 100 times faster system, which is why the project was called Hekaton, meaning 100 in Greek. [8]

Memory-optimized tables are stored in the main memory, providing substantial performance benefits [7]. They are completely durable and support most of the features that are supported by the disk-based tables, though there are some limitations [7]. Due to these limitations, Memory-Optimized tables cannot replace disk-based tables in all cases [7]. However, development is rapid and ongoing, so it would not be surprising if these limitations are removed soon. Databases can contain tables from both types, making it easier to integrate this technique into an existing database. Companies can choose to use memory-optimized tables only when a table is expected to handle many concurrent operations. However, integration can still be costly in some instances, and companies must evaluate the cost-benefit ratio. [8]

When storing data in memory, a large amount of memory is needed to support such tables [7]. Reserving more memory increases costs for the company [7]. Additionally, since it is a relatively new technique, developers require some time to adjust to it, which requires the effort of both the developers and the company. The company must assess the costs before utilizing such features in its database. Some companies

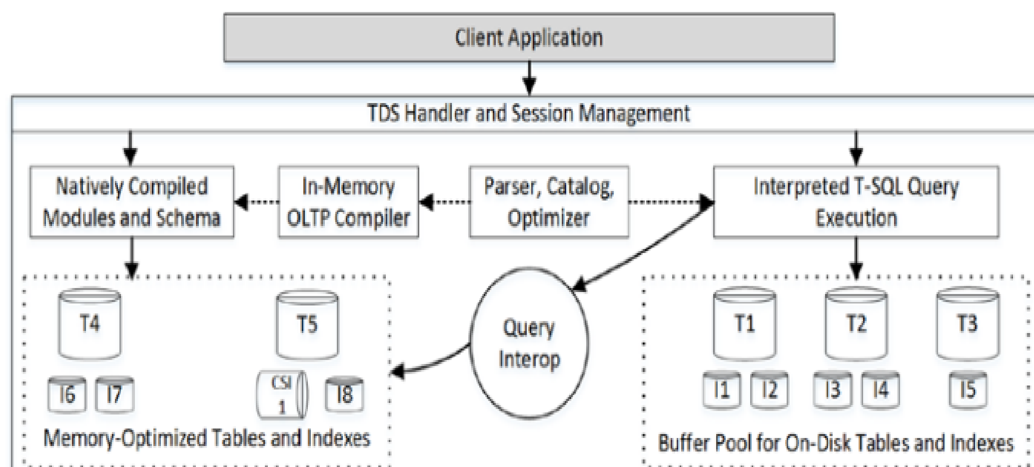

Figure 3.1 *Microsoft SQL Server Database Architecture [8]*

are willing to pay for this to achieve significant performance improvements.

Although both table types can be accessed through the well-known T-SQL querying language, they are implemented using different solutions. Differences can be seen in indexes, data storage, logging, data recovery plans, and garbage collection processes. [6][8]

3.1 In-Memory OLTP Architecture

Figure 3.1 shows where memory-optimized tables are located in the Microsoft SQL Server database architecture compared with the disk-based tables. Interpreted T-SQL accesses disk-based tables inside the buffer pool and also accesses memory-optimized tables through the query interop [8]. The location of the data is not visible to the querying language, which treats both types of tables the same way at a high level [8]. For example, execution plans, row accessing, and query optimization are performed regardless of which table type the query tries to access [8].

One objective of the new architecture is eliminating the need for latches and locks [4][6][21][23]. That has been achieved by using an optimistic approach [4][6][21][23]. Other objectives include optimizing data storage for main memory and utilizing native compilation [4][6][21][23]. Each of these targets will be studied in more detail, and how these have been taken into account in the architecture of the memory-optimized tables.

Begin Timestamp	End Timestamp	Index1 pointer	Index2 pointer	Column data

Figure 3.2 *Row structure in memory-optimized table [21]*

Latches and locks are bottlenecks when many concurrent sessions attempt to access and modify data [27][30]. These are necessary to protect the data within the data pages [27][30]. Memory-optimized tables use a latch-free architecture that utilizes an optimistic approach [6][10][17]. First, data pages have been removed, and data rows are linked using pointers [21]. Second, actions are gathered inside a transaction, and all the transactions start immediately once they occur, with the fastest one succeeding, and failed transactions will be restarted [21]. This is referred to as the optimistic approach because a transaction begins execution even if it attempts to modify data that is being modified by another ongoing transaction [21]. The fastest transaction commits its changes, while the other fails at the commit time [21]. This approach is made possible by the new multi-versioning technique that Microsoft has implemented in this feature [21].

The second objective is optimizing data storage for main memory [6][23]. That has been achieved by eliminating data pages, so the data no longer needs to follow the complex buffer pool structure and the code that manages it [6][23]. However, this approach has drawbacks. It can lead to slow database starts because indexes are not stored on disk and must be regenerated when a SQL Server instance is restarted. [4][12][21]

The third objective is to utilize native compilation [4][6][12][23]. This feature allows for the creation of natively compiled stored modules, which can be used when performance is critical [4]. These modules are compiled into fast C functions, which are then directly linked to the running Microsoft SQL Server service. This approach can lead to 20-40 times faster execution time in some cases, making it a powerful tool for boosting performance. [21]

3.2 Row structure and Multi Versioning Technique

Figure 3.2 illustrates the structure of a row stored in a memory-optimized table. Rows consist of five parts. The first part stores the timestamp at which the row is created. The second part stores the time at which the row is deleted. The remaining three parts are used to store pointers to the next row in the index row chain. Since the table can have multiple defined indexes, the row can participate in several index

chains. The last part contains the actual row data. [21]

The row contains two timestamp fields. The first value represents when the row was inserted, and the second value indicates when the row was deleted. This is how memory-optimized tables manage row multi-versioning. Each insert operation adds a new row and assigns the Begin Timestamp to the time at which the insert transaction begins. The End Timestamp part is assigned to ∞, signifying that the row is active and has not been deleted. During a delete operation, the End Timestamp is updated to reflect the time at which the delete transaction starts. An update operation is treated as one delete and one insert operation. First, the delete operation updates the End Timestamp of the old version, and then the insert operation adds the new version of the row. A row version is visible to transactions that have starting timestamps between the row's Begin Timestamp and End Timestamp. [8][10][21]

The actual deletion of a row occurs later as a part of the garbage collection background task. A row becomes visible to transactions once its End Timestamp is earlier than the starting timestamp of the oldest active transaction. During garbage collection, the row is first removed from the index row chains and then deleted. [8][10][21]

This means that memory-optimized tables can store multiple versions of the same row. Therefore, it is recommended to reserve enough memory to accommodate double the actual total table size. This behaviour should be considered while doing the cost calculations.

3.3 Indexes in Memory-Optimized tables

In Microsoft SQL Server 2016, memory-optimized tables should contain at least one index and a maximum of 8 indexes [7]. That limitation has been removed, and indexes can be more than 8 per table since Microsoft SQL Server 2017. One of those indexes in durable tables should be on the PRIMARY KEY [7]. Three different index types are available: hash indexes, non-clustered indexes, and column store indexes [6][7]. [8][21]

However, adding too many indexes can lead to performance issues. A high number of indexes can slow down row addition operations and increase memory usage. For this reason, tables should only include a reasonable number of indexes. [8]

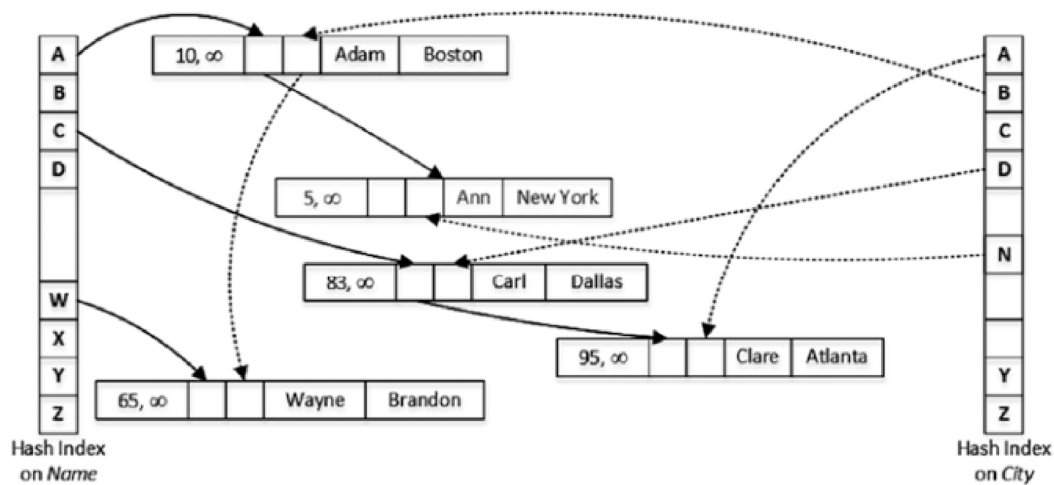

Figure 3.3 *Row binding using indexes*

3.3.1 Hash indexes

Hash indexes are designed for point look-up operations. It is beneficial when the query needs to check the equality of an indexed column to a specific value. [26]

When adding hash indexes to a table, it is essential to understand how they work. These indexes use a hash function to convert column values into hash snippets [26]. Values that produce the same hash snippet will be stored in one bucket [26]. Maintaining a small bucket size (the number of values that produce the same hash snippet) is important [26]. Otherwise, the performance will suffer. The decrease in performance is caused by how buckets are visited to find the value. The bucket stores 8 bytes that reference the first row in the bucket, and each row points to the next. If there are many items on one bucket, the engine must traverse many items to find the correct item in point-lookup actions. Conversely, having too many buckets means there will be many empty buckets, which will consume valuable system memory for no purpose [26]. It is recommended to target such a bucket count so that 33 percent of the buckets are empty. Moreover, it is safer to overestimate the number of buckets instead of underestimating it to avoid performance issues. [7][21]

As demonstrated in Figure 3.3, two example hash indexes are shown. The first one hashes rows according to the first name, and the other hashes rows according to the City Column. Each hashed snippet represents a bucket. That means if more than one row has the same name, both will end up in the same bucket. The bucket points to the first row, and each row points to the next. That explains why hashes are extremely important in memory-optimized tables. The number of buckets is also

essential because if there are too many buckets, many empty buckets will occupy precious memories. On the other hand, if there are not enough buckets, bucket size will grow, thus if we need to find one row that is located at the end of the bucket, many items will be traversed.

3.3.2 Non-clustered indexes

Non-clustered indexes in memory-optimized tables resemble those in disk-based tables but with key differences in how they function [7][8]. Indexes in disk-based tables utilize a B-tree structure to organize the data for various operations [8]. On the other hand, the non-clustered indexes of in-memory tables utilize the Bw-tree structure [8][12][16][26]. Bw-tree is latch free version of the popular B-tree [12][16]. At a high level, both work similarly, but non-clustered indexes cannot traverse the data backward [7][8]. That is why, during the design phase, the sorting order of the data should be considered [8].

When range searches (such as “<” and “>”) are needed, it is generally recommended to use non-clustered indexes. However, in lookup searches, the decision is not as straightforward due to how this index type works. Hash indexes enable point-lookup searches only when all index columns are present in the query condition. If the index is a combination of two columns and the query checks the equality of just one column, then the point-lookup search is not utilized, and the entire table data is scanned. In such situations, non-clustered indexes are the safer choice. Additionally, when data growth is unpredictable, it is advisable to use non-clustered indexes. When the amount of data grows dramatically, index chains in the hash index will grow, which makes point-lookup searches slower. [21]

3.3.3 Column-store indexes

The third index type supported by memory-optimized tables is column-store indexes. These are intended to speed up warehouse queries [19]. What are warehouse queries? Such queries process only a few columns but many rows [11]. The performance boost comes from storing the values of one column from different rows together [11]. However, in a regular MS SQL Server engine, the row data of different columns is stored together except for off-row columns [11][19].

Originally, the column-store index type was added to the MS SQL Server engine in the SQL Server 11 release, with a codebase named “Denali”. However, the first

version of memory-optimized tables in 2014 did not support column-store indexes. Support was added in SQL Server 2016. While column-store indexes are meant to make table scans more efficient, memory-optimized tables made lookup database searches faster. That is why a combination of these features seemed reasonable. [19]

3.4 Limitations

Even though Microsoft has tried to reduce the burden of using memory-optimized tables, there are still many essential features that are not supported. That is why, in some cases, memory-optimized tables cannot replace regular disk-based tables. It is worth mentioning that this feature is still under active development and may be removed in the future. The limitations, or in other words, the unsupported features of memory-optimized tables at the time of writing this thesis, are listed below with a brief explanation.

- Data compression for memory-optimized tables: This feature is used to reduce the size of the database by compressing the data. [21]
- Partitioning of memory-optimized tables, HASH indexes, and non-clustered indexes: Partitioning table and index data means storing the data of a table or index in more than one file group. [21]
- Replication: This feature is meant to move data from one database to another for backup, disaster recovery, reporting, or other purposes. Memory-optimized tables do not support replication, except for transactional replication to memory-optimized tables on subscribers. [21]
- Mirroring: Database mirroring involves having two identical instances of the database running on different SQL Server instances for better availability, increased data protection, and improved database availability during upgrades. [21]
- Rebuild log: This feature allows rebuilding logs through the use of the attach or ALTER DATABASE commands. [21]
- Linked Server: This feature gives the SQL Server database the ability to retrieve data from a remote database server that is running outside of the SQL Server instance. [21]

- Bulk logging: This is a recovery model. The recovery model determines how comprehensive the log is, whether log backup is needed, and what kinds of restore operations are available. In this model, log backups are required, and recovery points are at the end of any backup. No work is lost except for operations done after the last log backup. [21]
- Minimal logging: This involves logging only the information required to redo the transaction. However, storing minimal information does not allow point-in-time recovery. [21]
- Change tracking: When enabled for tables, this feature stores changes made to the table and allows them to be retrieved later for identification, such as row changes. [21]
- DDL triggers: These triggers are activated when Data Definition Language (DDL) events occur. DDL events include SQL statements that start with CREATE, ALTER, DROP, GRANT, DENY, REVOKE, or UPDATE STATISTICS. [21]
- Change Data Capture (CDC): When CDC is enabled, SQL Server Agent logs table insertions, updates, and deletions to make them easily accessible for consumption in a relational format. [21]
- Fiber mode: When fiber mode feature is enabled, context switching is done inline, reducing system overhead by minimizing kernel ring transitions. [21]
- Service Broker Limitation: The service broker brings the ability to connect multiple databases through queues. That helps divide the workloads across several databases without paying attention to the additional development complexity of the application. [21]
- Replication on subscribers: Memory-optimized tables can only be replicated when they are snapshot and transactional replication subscriber tables. [21]

Moreover, the following scenarios are not supported as well. Cross-database queries and transactions are not supported. If a transaction involves memory-optimized tables, it will not be able to access objects in other databases, with exceptions for tempdb, master, and resource databases. [21]

Using MERGE INTO statements on memory-optimized tables is not supported, which caused some additional work during the migration process from disk-based tables to memory-optimized ones. Event notifications, auto-close, derived columns,

policy-based management, and database containment are also not supported, which ultimately led to not migrating more than one table during the migration process. Additionally, foreign keys of memory-optimized tables are not allowed to refer to disk-based tables. Therefore, if all tables are connected through foreign keys, all tables need to be migrated. [21]

From the SQL Server data types list, memory-optimized tables do not accept datetimeoffset, hierarchid, sql_variant, geography, rowversion, XML, and geometry [7][3]. Memory-optimized tables cannot be accessed from inside CLR stored procedures using the context connection. Queries utilizing memory-optimized tables cannot utilize keyset or dynamic cursors. Finally, user transactions are not allowed to use transactional DDLs. [21]

3.5 Off-Row Data

In Microsoft SQL Server 2014, in-memory tables did not support off-row columns. For example, varchar(max) and varbinary(max) are considered off-row columns [8]. That significant limitation required schema changes to integrate disk-based tables into in-memory tables. This limitation was removed in Microsoft SQL Server 2016 [8]. However, the removal of this limitation comes with a cost.

Memory objects (consumers) send memory allocation requests to verheaps. Verheaps are data structures responsible for handling memory allocation requests. For example, when a new table is created with a primary key, a non-clustered index, and some in-row data, three memory objects are automatically created. The first one is for table data, and the other is for hash index (primary key) and non-clustered index. If we alter the table and add an off-row column, an additional three memory objects are allocated. One is for table data (off-row column), and the other is for range index heap and LOB Pace allocator.

As previously mentioned, memory-optimized tables store off-row data in a separate memory object, and the actual row uses an artificial key to reference its off-row data in other memory objects. This approach to removing the limitation causes additional memory allocations that are unnecessary in disk-based tables. However, this method decouples off-row and in-row data, meaning that if in-row data memory is modified, new versions of off-row data will not be generated, and vice versa. Off-row columns add additional overhead to the engine when accessing the data, which is why it is advised to avoid off-row data and convert it to in-row data before migrating disk-based tables to be memory-optimized tables. [8]

There is also a difference in how columns are stored in-row or off-row between memory-optimized and disk-based tables. In disk-based tables, the decision on where to place the data is made at runtime, while in memory-optimized tables, the decision is made when tables are created or altered. [3]

3.6 Table Altering

Starting from Microsoft SQL Server 2016, Memory-Optimized tables can be altered. However, that is not the same as in disk-based tables. It creates a new table with the additional changes and then copies the data from the old table to the new one. That is a background operation and is not visible to the users, which is why enough memory is required to contain multiple copies of the data. [3][8]

Table altering is a resource-intensive operation and may involve dropping the current table, creating a new one with the updated schema, and moving the data from the old table to the new one [3]. This heavy operation is performed offline and blocks access to the table rows during the operation [3]. There are two types of alterations: Metadata-only alteration and Regular alteration. Adding CHECK or FOREIGN KEY constraint is an example of the first type. In such changes, the table is not recreated; however, the engine may need to scan all data rows to ensure the constraint is valid. On the other hand, regular alterations require recreating a new table object with the updated schema changes and moving data from the old table to the new one. That is why those changes should be avoided whenever possible. [8]

3.7 Hardware requirements

Some hardware features should be considered to extract the most benefit from memory-optimized tables. The expected performance mainly depends on the number of CPUs in the system. Therefore, it is recommended to conduct tests and analyses to determine the suitable number of CPUs required now and in the future if data growth is anticipated. Using a higher clock speed instead of increasing the core count is also recommended, as it is a more cost-effective. Enabling simultaneous multithreading (SMT) with In-Memory OLTP tables is advised, as this can result in up to 40 percent faster execution times. Regarding system memory, it is advised to reserve enough to handle double the expected data size. However, additional steps should be taken to ensure sufficient memory is available for future data growth and

to reserve additional memory for the server as needed. An out-of-memory situation will make the data read-only until either the amount of data is reduced or more memory is added. [8]

To use In-Memory, no updates, installations, or other configurations are needed on Microsoft SQL Server. Everything comes with the Microsoft SQL Server, which is why it is crucial to keep the table compliance level consistent with the server version. This technology is rapidly evolving, and new features are introduced regularly. It is recommended to update the Microsoft SQL Server version consistently. Memory-optimized data is stored in a separate file group. File groups should be optimized for sequential reads because In-Memory OLTP streams are based on sequential I/O access instead of random I/O access. Sequential append-only writes to be used when the server is running, and sequential reads are used during database startup and recovery stages. Once the file group is created, it can be deleted if nothing has been stored there. However, after the first file is stored, the file group cannot be removed, even if everything has been deleted. These behaviours should be considered when designing Memory-Optimized tables. [8]

3.8 Customer Cases

In this section, we will explore two examples of how memory-optimized tables helped companies achieve better performance.

The first customer case is related to IoT (internet of Things). The company in this case sells IoT devices that their customers use daily in their homes. These devices send diagnostic data, which is stored in an Azure SQL Database. On average, 750 million rows are inserted daily into over 70 tables in the database. The issue was that the database reached its log I/O limit for the pricing tier. Since the database did not use any features that would block the migration to memory-optimized tables, the company only needed to change the queries that created the tables. After migrating the tables, they handled 30-40% more data. [20]

The customer of the second customer case is bwin. Bwin is one of the leading betting companies in Europe and has offices in many countries worldwide. Their platforms receive more than 19 million bets daily, so they need to stay up-to-date with the latest technologies to achieve better performance. Bwin used distributed cache-based systems like Cassandra, Memcached, and Microsoft AppFabric to handle that magnificent number of transactions. These cache systems suffered from stability issues

due to the workload. Adopting the in-memory OLTP engine was straightforward, as the only change required was updating a single DLL. Nevertheless,performance increased significantly. Using a mid-tier cache solution with SQL Server, the performance was 150,000 batch requests/sec per node, while after using the In-memory OLTP of SQL Server, the performance increased to 1,200,000 batch requests/sec per node. That also allowed them to reduce the number of nodes required from 19 nodes to one. They achieved such performance gain gradually. First, with the SQL Server 2014 version, the performance was 300,000 batch requests/sec per node because of the large limitation this feature had at that time. First, they used it in their ASP.Net SessionState component, and after they saw the gains, they gradually replaced other distributed cache systems. [19]

Regardless of previously mentioned customer cases, this feature is not suitable for each situation, so it is recommended to make a comprehensive study beforehand to identify possible blockers and measure the gains and costs.

4 M-Files Vault Database

M-Files is a software company specializing in enterprise information management. It utilizes the M-Files application to bring value to its customers by digitizing their processes. Below, the main features of the application are covered. The core building block of the application is a vault. The vault can be considered a root-level container where objects can be placed. Objects uploaded to the application automatically become version-controlled. That allows users to view the object's history, revert to an older version, and check the author of a specific version. Users can also add predefined workflows to objects. This feature enables companies to automate their processes and control the workflows of different object types. For example, objects can be documents, assignments, customers, emails, reports, etc. Additionally, other special object types can be created for M-Files customers. [22]

One of the key features of the M-Files application is object metadata. For each document, you can add different key-value pairs that describe it(see Figure 4.1 below for context). The document can be searched using the properties defined in its metadata. This enhances the way users manage documents and search for them. That would be a handy feature for companies that struggle to find the correct document efficiently. [22]

Due to the application's size and its support for many features, the vault database is quite large, containing, in some cases, up to 500 database tables. Some tables are accessed frequently, while others are only accessed during application restarts. The change tracking feature is enabled for only a few tables. A minor part of the tables has no relationships with other tables. The database structure does not follow the best database modelling practices for performance reasons. For example, the same information may be stored in different tables. This was intentionally done to enhance search operations. All vault database tables are disk-based.

This research studys whether we can transfer all those tables to the main memory. The database uses Firebird or Microsoft SQL Server, which is why, in this study, we focus on the In-memory OLTP engine and especially on the memory-optimized tables provided by the Microsoft SQL Server.

In the later part of this section, we will use the Transaction Performance Analysis

Figure 4.1 Metadata Card

Overview report to identify which tables could potentially benefit from performance improvements. Finally, we will walk through the process of migrating the tables and the changes made to the M-Files server.

4.1 M-Files Vault Operations

This subsection dives into the operations whose performance are tested. These operations are performed daily using M-Files software. The first operation is object creation. When a user creates a new object, descriptive information should be assigned. This information will be used to find the object more easily. In M-Files

terminology, the assigned information is called metadata. Metadata can include name, time created, time modified, created by, etc. The user can assign some of this information, and some is generated automatically.

The second operation is metadata modification. As explained, each object has its own metadata that can be assigned during its creation. However, object metadata can also be changed after the object has been created. Metadata modification means adding new, removing old, or modifying an existing value. In figure 4.1, the metadata card of a document can be seen. It shows that the document has the name of "Thesis plan", the class of "Document" and much other information.

The third operation is check-in. Before the user can modify a document, it should be checked out. This will prevent other users from modifying the document at the same time. When the user is ready to publish the changes made, the check-in operation should be performed. This will commit to the new version of the document. The document in figure 4.1 is checked out to Ameer Kulabi, which means that he is the only one who can modify this document now.

The last operation is search. Because there is metadata attached, those can be used to search for the objects. The more descriptive the metadata, the more easily it can be found through searches. For example, the users can search for all objects modified by them in the past 30 days. Searches play an important role in building a smooth user experience by minimizing the time required to find a document.

4.2 Transaction Performance Analysis Overview

To understand the current situation better, SQL Server Management Studio provides built-in tools to extract important reports. These tools were used on M-Files' internal vault, called Motive. There are other internal vaults inside M-Files, but this one was selected because it is the most actively used vault among them. The reports below have been generated from the database of that vault.

In figure 4.3, we can see that most of the tables are concentrated in the bottom-right corner, which is a group of tables with minimal migration load and low-performance gain. We can also notice that the EVENTRECORD table is the highest on the y-axis, indicating that it will benefit the most from this migration. However, EVENTRECORD affects the figure by pushing more important tables down along the y-axis. More important tables are considered to be related to documents, while the

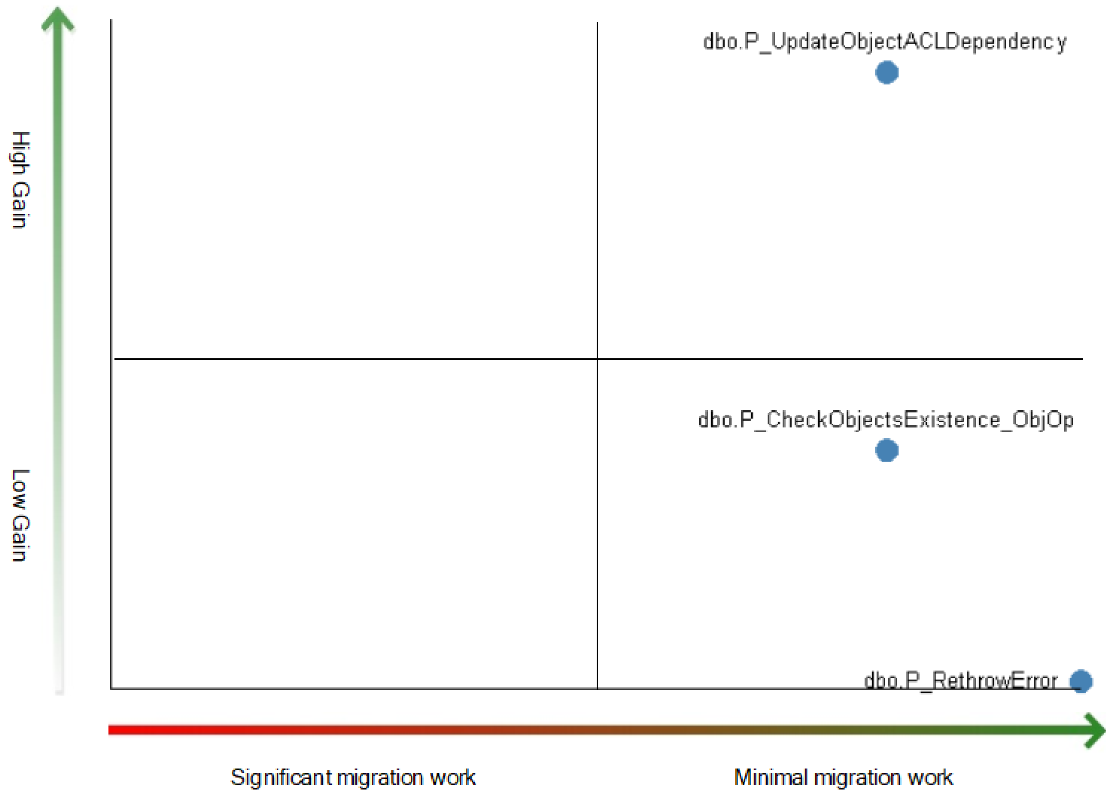

Figure 4.2 *M-Files Vault database procedures.*

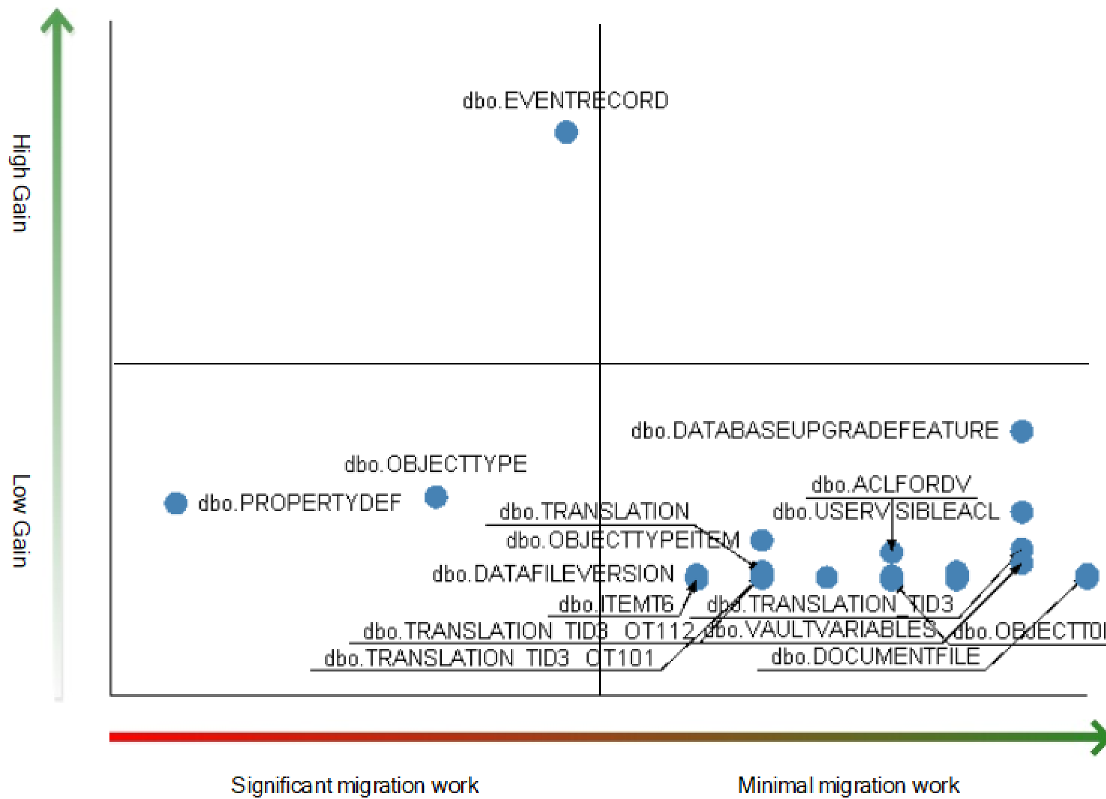

Figure 4.3 *M-Files Vault database tables.*

EVENTRECORD table is meant to store user activity for audit trial purposes. We can also see the PROPERTYDEF and OBJECTTYPE tables in the bottom left corner. Even those are hard to migrate due to the existing limitations, both are extensively used tables in document related queries and worth migrate.

All M-Files procedures are shown in figure 4.2. We will not go into much detail as it is outside the scope of this study. However, P-UpdateObjectCLDependency shows the potential to benefit from the native stored procedures feature.

4.3 Implementing changes

This section explains the database migration process step by step and demonstrates all the software changes required. This is important for the company if it decides to continue the implementation because it will help the developers understand what has been done and what is still needed. They can also follow the same migration process, which is explained later in this section. Before we proceed with the migration process, let us see what made the migration possible. First, all foreign keys were removed because, as mentioned earlier, there cannot be a foreign key connection between memory-optimized and disk-based tables. Foreign keys were also removed from the original database. Furthermore, triggers were removed from both memory-optimized and disk-based tables, as triggers are not supported in memory-optimized tables. In this way, we tried to keep a similar schema in both databases to achieve the most reliable results possible. We migrated most of the database tables during the migration process. However, some tables were not migrated due to feature limitations. The isolation level of memory-optimized tables was set to "snapshot," which is compatible with the isolation level of disk-based tables. Check the allowed isolation levels in figure 2.1.

The migration tool allows the user to select the type of indexes for the migrated tables. There are two types of indexes to choose from: non-clustered indexes and hash indexes. That is why it is essential to identify the use cases of the table and the queries in which it is used. If all index columns are used as a filter in a query with the look-up searches, it is better to select the hash index type. When this type is selected, it is essential also to check the number of buckets needed because a small value could lead to a significant performance decrease. On the other hand, a large value leads to allocating a significant amount of unnecessary memory. However, if these conditions are not met, choosing a non-clustered index, which behaves like a non-clustered index in disk-based tables, is safer. When this type is selected, it

is important to observe the use cases of the table, as the migrator should choose a sorting order that is less bidirectional than the disk-based non-clustered index. Column-store indexes remain the same in the migrated tables because memory-optimized tables also support these indexes.

Moreover, there is an option to select whether to copy the existing data from the disk-based table. During the migration process, only the data in tables that contain necessary information for the vault's function is copied. For example, we copied built-in object types and classes. Below, how the tables have been migrated is discussed.

All search structure property-based tables, or the so-called OVPV tables, were migrated with all indexes chosen as non-clustered indexes, while the primary key was set as a hash index. This configuration will make look-up searches faster. All other indexes are non-clustered indexes. Column-store indexes remained the same because both tables support this index type. The primary key of other tables was selected as a non-clustered index because it showed better performance in the results. This is because changing all the queries to benefit from hash indexes is challenging due to the structure of the queries. See Section 3 for more information about this type of index and what is needed to make the query benefit from this index type.

Next, the changes made to the M-Files software server are explained. We split MERGE INTO queries into multiple queries because memory-optimized tables do not support MERGE INTO queries. This led to an increased number of queries in one API CALL. Still, because memory-optimized tables intend to perform better, API calls generally perform better with memory-optimized tables.

The widely used table ACLFORDV was not migrated because it contained a column called HASH, which is automatically calculated using the built-in function binary_checksum(). We believe that this table could affect the result positively once migrated, as it is utilized in many queries. Moreover, we have not migrated the EVENTRECORD table because one of its columns is of type XML, which is not supported in memory-optimized tables. To eliminate the impact of this on the results, we removed the codebase that interacts with this table. Other tables were not migrated for similar reasons.

Tables were migrated in batches, and after each batch, the server was adjusted to be compatible with the database. This approach is easier because it allows us to approach each problem individually, and it also gives us more visibility into how

each batch affects the results. We have not listed the results after each batch, only before and after migrating all the tables.

5 Method of research

M-Files aims to handle larger data smoothly. This objective creates the need for faster applications capable of processing vast amounts of data in a reasonable time. One way to achieve faster applications is by optimizing the database as efficiently as possible. Memory-optimized tables appeared to fulfill those requirements. However, there was considerable uncertainty about whether this feature would suit the current M-Files Vault database and how much performance gain it would provide.

First, the necessary database changes are implemented to ensure the feature is appropriate. In the second phase, two PowerShell scripts were implemented. The purpose of this script is to send requests to the server and record the time taken for the server to respond. The first script sends object creation, object check-in, and object modification requests to the server, while the second script sends search operation requests. Both work in this way: first, the script sends one request and measures the time taken for the server to replay. Then, it sends two requests, and measures the collapsed time, after that it sends three requests, etc. The script continues until the time of 4500 objects is measured. This way, we can compare how much slower the database becomes as its size increases. Simultaneously, we can compare the current application results with the migrated one. It was decided that the script would only measure the throughput time of the API request to simulate the benefit seen by the end-user, which is the primary objective that initiated this study.

The scripts were run on the same computer that hosts the M-Files server to minimize the effect of GRPC requests on the results. Moreover, for the same reason, we have ensured that the computer does not have any other running programs that could affect the running scripts and the performance of the main memory, which affects the performance of memory-optimized tables. The computer used for the tests had 64 GB of main memory and a 128-core processor.

To eliminate potential errors, the scripts were run up to 4 times, and the average was taken as the result. The company provided technical support and reviewed the implementation to ensure that it was sufficient for its needs.

The server code used and generated results were permanently stored in the M-Files

version control platform so we could redo the experiment at any time in the future. If the company implements the solution after further investigation, it could also be a starting point for the actual implementation.

The test setup is simple. The scripts described earlier in this section are run in Powershell. Both use the GRPC protocol to send requests to MFServer, a Windows service running on the same computer that runs the scripts. Each of the examined M-Files vault operations mentioned in Subsection 4.1 contains tens of SQL queries run in the SQL Server instance that contains the vault database. In addition to the queries, in MFServer, there is code that manipulates the results in the required format. This study focuses on the performance of the queries. Still, it was decided to study the performance of the M-Files vault operations in general, as the target of the study is to measure the benefits seen by the users of the application. There were two versions of MFServer. One was for the disk-based vault; this version is similar to the currently used M-Files vault with limited changes. The second version was used for the memory-optimized vault, customized to suit the underlying memory-optimized tables. In the memory-optimized vault, a SERIALIZABLE isolation level was used for disk-based tables and SNAPSHOT for memory-optimized tables. In the disk-based vault, the SNAPSHOT isolation level was used. Examples of the run queries can be seen in Figures 5.1 and 5.2. In the first query, object information is searched from three different tables, and this operation is executed in check-in, object modification, and object creation operations. On the other hand, the second query calculates the physical file size of an object, and it is executed within the check-in operation.

```sql
SELECT /* Q: GetObjectDBStructureAspectsForObjects SingleObject */
    CAST( 1 AS INTEGER ) AS TableID,
    CAST( 0 AS INTEGER ),
    ID,
    CASE WHEN LatestCheckedInVersion IS NULL THEN -1 ELSE LatestCheckedInVersion END AS LCI,
    CASE WHEN RemovedAt IS NULL THEN 0 ELSE 1 END AS Removed
FROM OBJECTT0R
WHERE ID = ? UNION ALL

SELECT
    CAST( 2 AS INTEGER ) AS TableID,
    CAST( 0 AS INTEGER ),
    ID,
    CASE WHEN LatestCheckedInVersion IS NULL THEN -1 ELSE LatestCheckedInVersion END AS LCI,
    CASE WHEN RemovedAt IS NULL THEN 0 ELSE 1 END AS Removed
FROM OBJECTT0I
WHERE ID = ? UNION ALL

SELECT
    CAST( 3 AS INTEGER ) AS TableID,
    CAST( 0 AS INTEGER ),
    ID,
    CASE WHEN LatestCheckedInVersion IS NULL THEN -1 ELSE LatestCheckedInVersion END AS LCI,
    CASE WHEN RemovedAt IS NULL THEN 0 ELSE 1 END AS Removed
FROM OBJECTT0IS
WHERE ID = ?
```

Figure 5.1 *Example query used in many examined operations.*

```sql
SELECT SUM( DFV.PhysicalFileSize )
FROM DV_DF A
    INNER JOIN V_DocumentFileVersion DFV
        ON DFV.ID_DocumentFilePart = A.DFV_DocumentFilePart AND
        DFV.ID_VersionPart = A.DFV_VersionPart
WHERE A.DFV_DocumentFilePart <> 0 AND
    A.DV_ObjectType = ? AND
    A.DV_DocumentPart = ? AND
    A.DV_VersionPart = ? AND
    NOT EXISTS (
            SELECT *
            FROM DV_DF B
                INNER JOIN DocumentFileVersion DFV_B ON DFV_B.ID_DocumentFilePart = B.DFV_DocumentFilePart AND DFV_B.ID_VersionPart = B.DFV_VersionPart
            WHERE DFV_B.ID_DocumentFilePart = DFV.ID_DocumentFilePart AND DFV_B.DataFileVersion = DFV.DataFileVersion AND B.DV_VersionPart < A.DV_VersionPart )
```

Figure 5.2 *Example query used in many examined operations*

6 Results and performance comparison

Recorded data samples were stored in an Excel worksheet. Those sheets were then used to draw a figure comparing the old and new system. We have added a separate figure for each operation, or in other words, for each API call.

Figure 6.1 shows the result of the check-in operation. The horizontal axis represents the number of API calls or the number of processed objects. The time spent creating those objects is shown on the vertical axis. In Figure 6.2, the performance of the metadata changes is displayed. For metadata changes, we have used the setProperties API method. Figure 6.3 demonstrates the performance of the object creation operation using the CreateNewObject API method. The last two figures, 6.4 and 6.5, show the performance data for search operations.

Before running the scripts, both databases were empty. After execution, the databases contained 10 million objects. The disk-based vault allocated 50 GB of disk space, while the memory-optimized vault allocated 55 GB from the main memory. Most of the memory was allocated by tables and indexes. Upon depleting the available RAM, the memory-optimized tables began to perform poorly, which is expected behavior in such situations, as explained in Section 3.

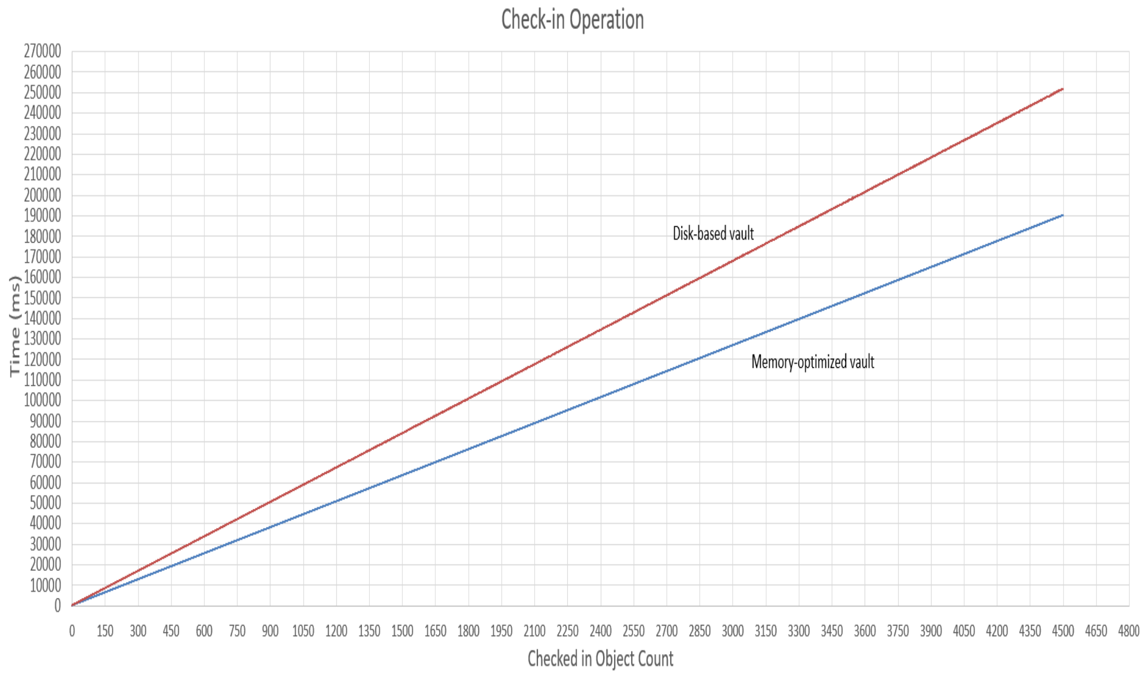

Figure 6.1 *Check-in operation performance comparison.*

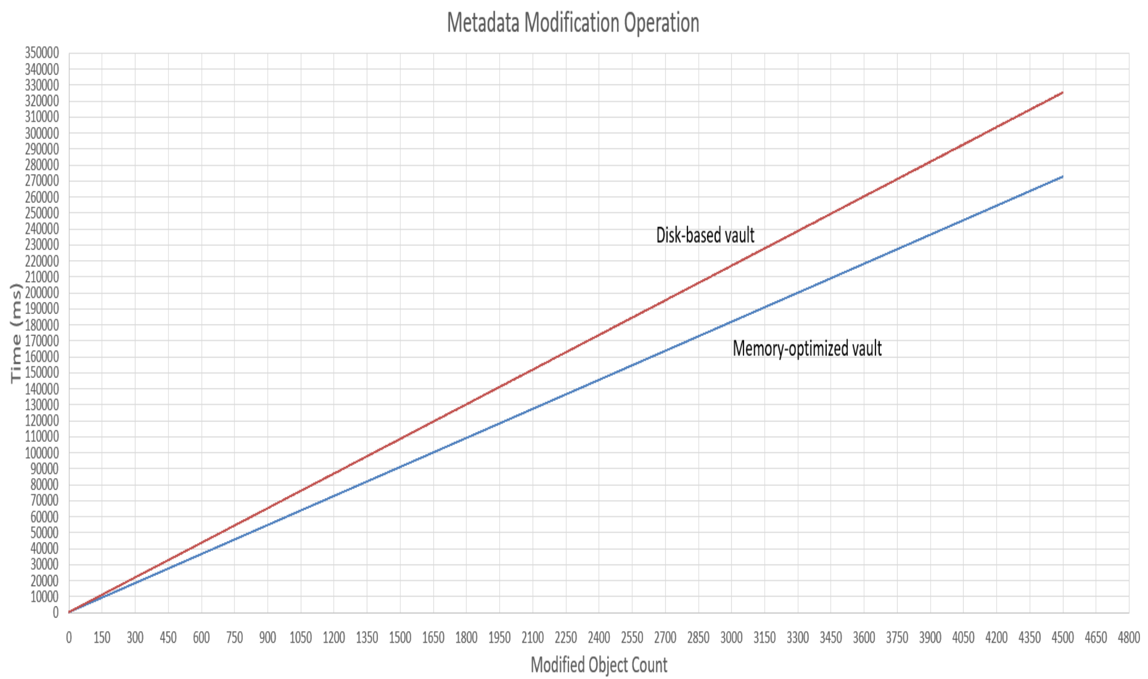

Figure 6.2 *Metadata change operation performance comparison.*

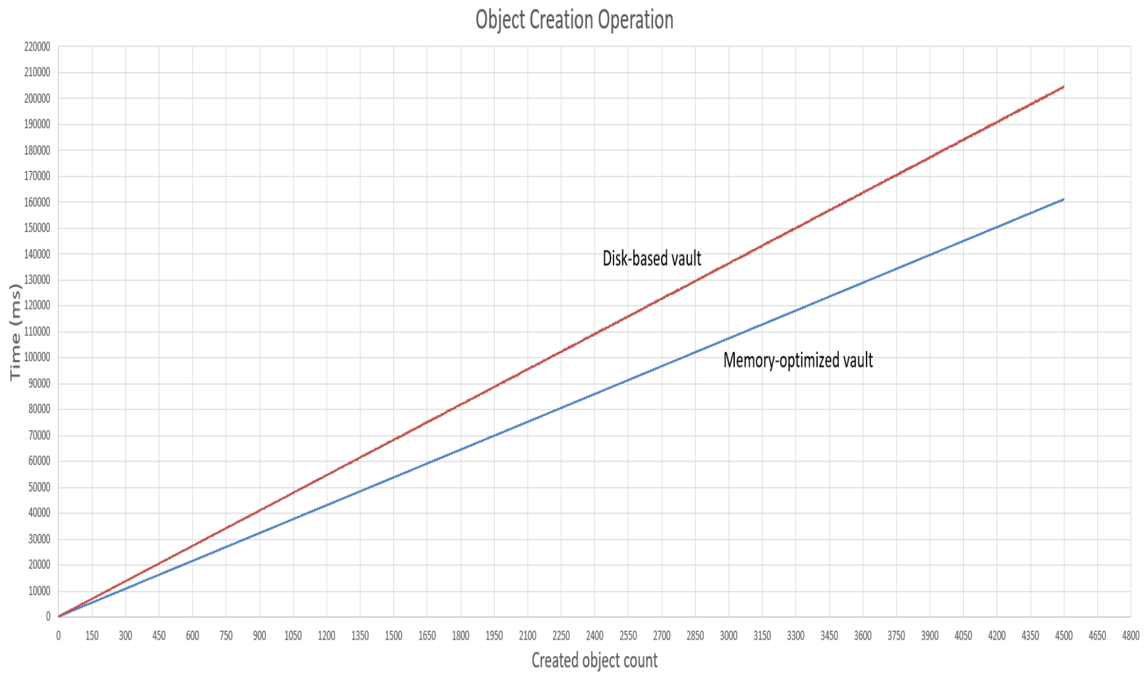

Figure 6.3 *Object creation operation performance comparison.*

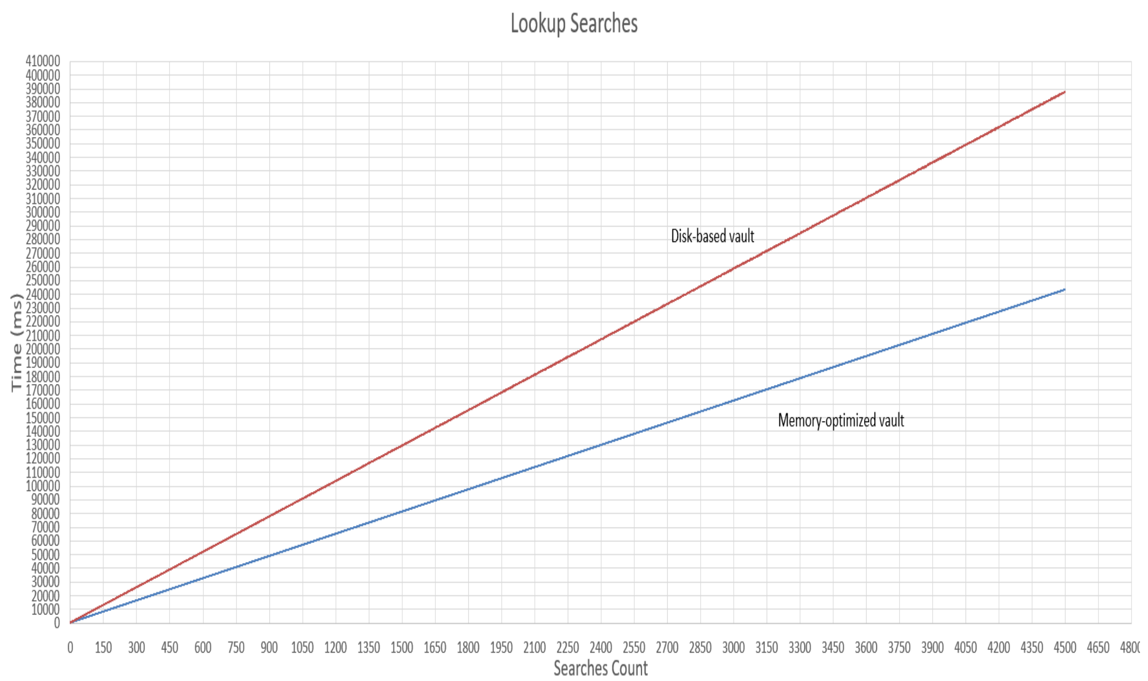

Figure 6.4 *Lookup search operation performance*

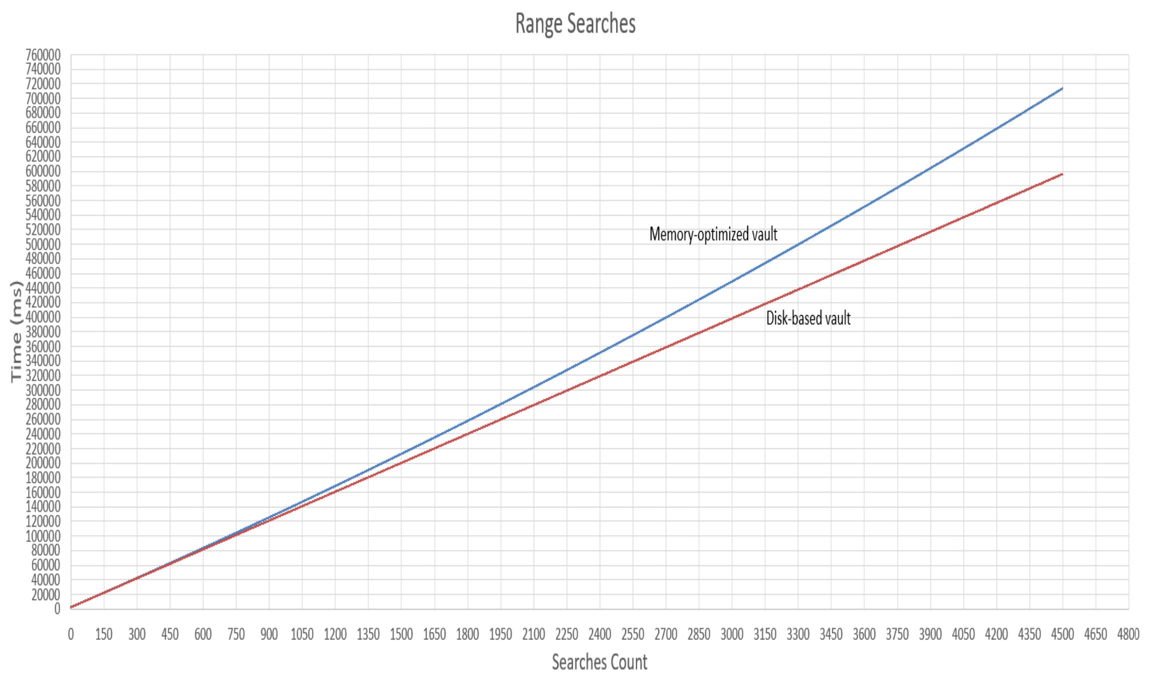

Figure 6.5 *Range search operation performance*

7 Discussion

As we can see, in almost all figures, memory-optimized tables performed better than disk-based tables, as expected. The time spent shows linear growth as the number of handled objects increased. However, this is not the case with Figure 6.5, which shows the data of range searches. This behaviour occurs because we used the hashed index as the primary key index type. As explained in previous sections, this index type enhances look-up operations but at the cost of slowing down range operations. Full table scans were used because no other indexes could be used.

In addition to the performance comparison, the size of the memory-optimized tables should also be considered. A 10 million object vault may require more than 100 GB of RAM, which introduces additional costs. Because memory-optimized tables enhance concurrent operations, purchasing faster processing units could further enhance overall performance.

Although most operations showed performance gains, the difference between the migrated and original systems was not as significant as expected. Not all tables have improved performance due to their structure, and some tables were even slower. The factors that made certain tables slower are explained below.

One factor that impacted the results was the off-row data used in many tables. As mentioned in Section 3, memory-optimized tables store off-row data in a separate memory object from in-row data, and SQL Server uses a pointer to bind the two. This leads to slower performance than could potentially be achieved.

Additionally, the index structure needs to be redesigned to achieve better performance. As the results show in Section 5, it was managed to fasten all operations except range searches. The reason for this is the current structure of indexes in the vault database. To better understand why the current index structure is unsuitable, see Figures 7.1 and 7.2. Those figures show the query plan of the queries shown in Section 5. Because of differences in how those indexes behave in memory-optimized tables and disk-based tables, it can be seen that table scans were used instead of index scans. This caused the query to be slower, as seen in Figure 7.1. On the contrary, in Figure 7.2, it can be seen that a faster query plan was generated in some queries. For example, in that figure, the clustered index of the disk-based tables was

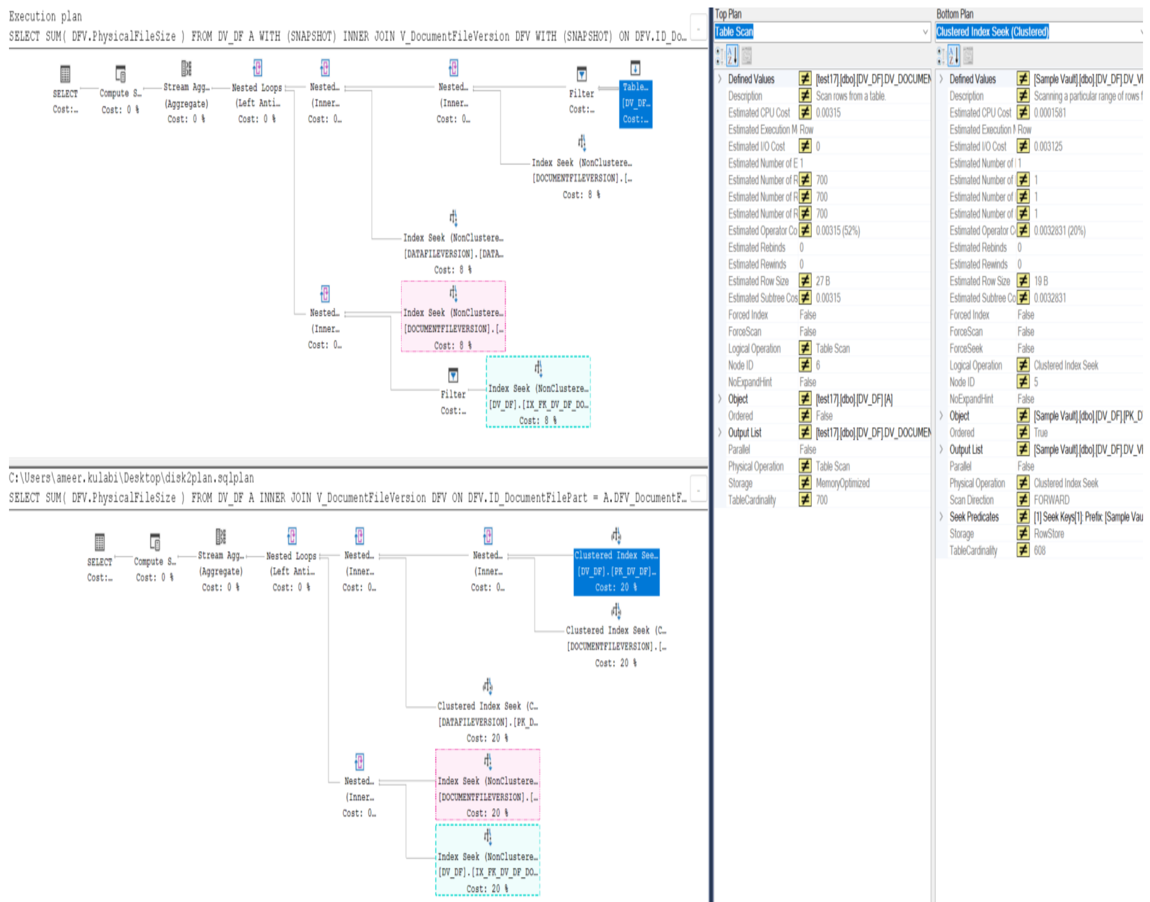

Figure 7.1 *Execution plans for query displayed in Figure 5.2 in disk-based and memory-optimized vault*

replaced with the non-clustered hash index of the memory-optimized tables. The second one is faster with look-up seeks as it needs less than 1000 instructions to find a record [12]. Each examined M-Files vault operation becomes either faster or slower depending on the percentage of queries that become slower or faster. Because of this, it can be seen from Figures 6.1, 6.2, and 6.3 that those obtained similar performance benefits because they execute similar queries.

Because memory-optimized tables do not support all data types as disk-based tables, the design of some tables, such as EVENTRECORD and ACLFORDV, should be reconsidered. Furthermore, existing triggers must be implemented in code because they are not supported in memory-optimized tables. Additionally, the Change tracking feature is unavailable for memory-optimized tables, which may add more complexity to the code of the M-Files application server once implemented.

In my opinion, this feature is not yet suitable for the M-Files Vault database due to the limitations mentioned or because it is not suited for the entire database. When

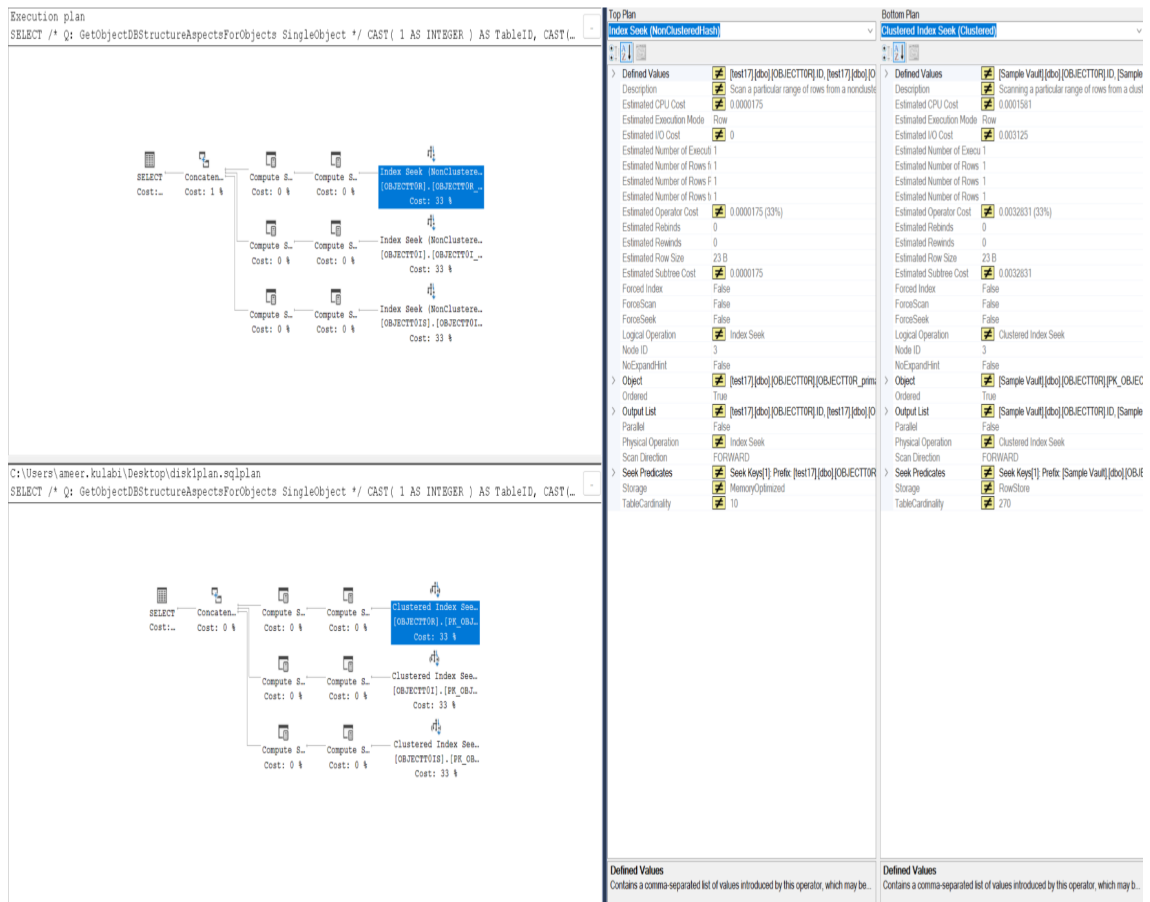

***Figure* 7.2** *Execution plans for query displayed in Figure 5.1 in disk-based and memory-optimized vault*

we compare the results from this study to those obtained by other companies, we notice that, in their cases, the only necessary changes were how tables were created and the preparation for additional memory. Also, they did not migrate the whole database. They migrated only those tables that needed performance improvements or suffered from large lock and latch wait times. Ayub M. B.* and Ali N. compared in-memory and disk-based tables [1]. In the comparison, they made the database faster in all their use cases [1]. However, their setup is different from that of this thesis. In their case, the database was simple, with only a few tables [1]. They were also able to redesign the indexes, making all use cases faster [1]. In this thesis case, the database is more complex, and thus changing the index structure requires additional study.

If M-Files wishes to use this memory-optimized tables feature now, they could migrate only certain tables. For example, tables that do not have foreign key relationships with other tables, do not have triggers and do not use the Change Tracking feature could be migrated without any changes to the server, aside from how those

tables are created. Alternatively, M-Files could redesign parts of the database that require performance improvements and migrate only those sections.

Finally, I would like to state that the setup used to conduct these tests has limitations, like any other performance comparison study. For example, we had to remove foreign keys and triggers from the setup due to their limitations. Additionally, the available memory and processing units and their performance differ from those used in the computers that run the servers. Adding more cores and to the host machine could effect on the performance significantly [12].

8 Conclusion

Memory-optimized tables are a relatively new database table type in SQL Server, which was introduced in 2014 by Microsoft. The Data of these tables is stored entirely in the main memory. This means that whenever the data is modified or accessed, it does not need to be retrieved from the disk, as it is already available in the main memory. When this table type was developed, three primary goals were considered. The first was to eliminate latches and locks, the second was to remove data pages and optimize data storage for main memory, and the third was to utilize native compilation with procedures.

Unfortunately, this feature comes with limitations, which make it not suitable for every case and cannot replace disk-based tables in every situation. For example, memory-optimized tables do not support all data types like disk-based tables. Additionally, change tracking and triggers are not supported by this new table type. Limitations are explained in more detail in Subsection 3.4.

In Section 4, we introduced M-Files, the case company. M-Files is a software company specializing in enterprise information management. They wanted to test the suitability of the memory-optimized tables to their M-Files vault database. To evaluate how well this feature would suit their database, we migrated most of the tables in the database to be memory-optimized. Along the way, we encountered obstacles that could hinder the integration of this feature. Those obstacles are making the support of change tracking, triggers, and some data types.

After completing the migration, we implemented two scripts for benchmarking purposes. The objective of these scripts was to measure how fast the database with memory-optimized tables is. Additionally, they will also measure if the database performance will be decreased when we have a huge amount of objects in the tables. It was decided to measure the performance of the object check-in, the object creation, the object metadata modification, the range search, and the lookup search operations. The performance comparison in Section 6 demonstrated that most of the studied operations were faster with memory-optimized tables.

In general, this feature is more suitable for databases that do not rely on unsupported features and that experience significant latch and lock wait times. Additionally,

future data growth should be taken into account, as enough memory should be reserved to handle double the actual size of the data. This is critical because if there is insufficient memory, the database will enter read-only mode. This issue can be resolved by adding more memory to the system.

If M-Files decides not to start utilizing memory-optimized tables in their database because of the existing limitations, it is worth following this feature, as it may become more feasible once those limitations are addressed. On the other hand, if the decision is made to utilize this feature, the following steps would be finding ways to overcome the limitations by implementing similar functionalities in the M-Files application server.

9 References

[1] Ayub, M. B., & Ali, N. (2018) Performance comparison of in-memory and disk-based databases using transaction processing performance council (TPC) benchmarking. Journal of internet and information systems. [Online] 8 (1), 1–8.

[2] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (1995) 'A critique of ANSI SQL isolation levels', in Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. [Online]

[3] Dejan, S., Milos, R., & William, D., (2017) SQL Server 2016 Developer's Guide. Packt Publishing.

[4] Diaconu, C., Freedman, C., Ismert, E., Larson, P.-A., Mittal, P., Stonecipher, R., Verma, N., & Zwilling, M. (2013) 'Hekaton: Microsoft SQL Server's memory-optimized OLTP engine', in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. [Online]. 2013 ACM. pp. 1243–1254.

[5] 'Exploring the Firebird Database' (2023) Open Source FOR You, 09 Aug, NA, available:
`https://link-gale-com.libproxy.tuni.fi/apps/doc/A760248500/ITOF?u=tampere&sid=bookmark-ITOF&xid=6554a675`

[6] Faerber, F., Kemper, A., Larson, P. Å., Levandoski, J., Neumann, T., & Pavlo, A. (2017) Main Memory Database Systems. Foundations and trends in databases. [Online] 8 (1–2), 1–130.

[7] Fritchey, G. (2018) 'Memory-Optimized OLTP Tables and Procedures', in SQL Server 2017 Query Performance Tuning. [Online]. United States: Apress L. P. pp. 753–781.

[8] Korotkevitch, D. (2017) Expert SQL Server in-Memory OLTP. 2nd ed. edition. [Online]. Berkeley, CA: Apress L. P.

[9] Lahiri, T., Neimat, M. A., & Folkman, S. (2013). Oracle TimesTen: An In-Memory Database for Enterprise Applications. IEEE Data Engineering Bulletin,

36(2), 6-13.

[10] Larson, P.-Å., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., & Zwilling, M. (2011) High-performance concurrency control mechanisms for main-memory databases. Proceedings of the VLDB Endowment. [Online] 5 (4), 298–309.

[11] Larson, P.-Å., Clinciu, C., Hanson, E.N., Oks, A., Price, S.L., Rangarajan, S., Surna, A., & Zhou, Q. (2011) 'SQL server column store indexes', in Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. [Online]. 2011 New York, NY, USA: ACM. pp. 1177–1184.

[12] Larson, P.-Å., Hanson, E. N., & Zwilling, M. (2015) 'Evolving the architecture of SQL Server for modern hardware trends', in 2015 IEEE 31st International Conference on Data Engineering. [Online]. 2015 IEEE. pp. 1239–1245.

[13] Larson, P.-Å. & Levandoski, J. (2016) Modern main-memory database systems. Proceedings of the VLDB Endowment. [Online] 9 (13), 1609–1610.

[14] Larson, P. Å., Zwilling, M., & Farlee, K. (2013). The Hekaton Memory-Optimized OLTP Engine. IEEE Data Engineering Bulletin, 36(2), 34-40.

[15] Lee, J., Muehle, M., May, N., Faerber, F., Sikka, V., Plattner, H., Krueger, J. & Grund, M. (2013). High-Performance Transaction Processing in SAP HANA. IEEE Data Engineering Bulletin, 36(2), 28-33.

[16] Levandoski, J., Lomet, D., Sengupta, S., Birka, A., & Diaconu, C. (2014) 'Indexing on modern hardware: hekaton and beyond', in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. [Online]. 2014 New York, NY, USA: ACM. pp. 717–720.

[17] Lomet, D. (2013) Microsoft SQL server's integrated database approach for modern applications and hardware. Proceedings of the VLDB Endowment. [Online] 6 (11), 1178–1179.

[18] McQuillan, M. (2015) Introducing SQL Server. 1st ed. 2015. [Online]. Berkeley, CA: Apress.

[19] Microsoft Learn. (2024) Bwin's exprience of utelizing memory-optimized tables. [Website]

https://learn.microsoft.com/fi-fi/archive/blogs/sqlcat/how-bwin-is-using-sql-ser

[20] Microsoft Learn. (2024) IoT company's experience of utelizing memory-optimized tables. [Website]
https://learn.microsoft.com/en-us/archive/blogs/sqlserverstorageengine/a-technical-case-study-high-speed-iot-data-ingestion-using-in-memory-oltp-in-azu

[21] Microsoft Learn. (2024) SQL Server technical documentation. [Website]
https://learn.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver16

[22] M-Files User Guide. (2024) Official user guide. [Website]
https://userguide.m-files.com/user-guide/latest/fin/intro_to_m-files.html

[23] Nevarez, B. (2014) 'In-Memory OLTP aka Hekaton', in Microsoft SQL Server 2014 Query Tuning and Optimization. United States: McGraw-Hill Education. p.

[24] Oracle Documentation. (2024) Oracle TimesTen In-Memory Database Documentation. [Website]
https://docs.oracle.com/en/database/other-databases/timesten/22.1/

[25] Petkovic, D. (2020) 'Part II Transact-SQL Language', in Microsoft SQL Server 2019: a Beginner's Guide, Seventh Edition. United States: McGraw-Hill Education. p.

[26] Pollack, E., & Strate, J. (2023) 'Indexing Memory-Optimized Tables', in Expert Performance Indexing in Azure SQL and SQL Server 2022. [Online]. United States: Apress L. P. pp. 201–215.

[27] Ryan, J., Ippokratis, P., Nikos, H., Anastasia, A., & Babak, F. (2009) Shore-MT: a scalable storage manager for the multicore era. EDBT 2009: 24-35

[28] SAP HANA Platform. (2023) SAP HANA Developer Guide For SAP HANA Studio. [Online]
https://help.sap.com/doc/fbb802faa34440b39a5b6e3814c6d3b5/2.0.07/en-US/SAP_HANA_Developer_Guide_for_SAP_HANA_Studio_en.pdf

[29] Savorgnano, S. (2014) SQL Server 2014's Analysis, Migrate, and Report Tool. SQL Server Pro, 16(4), pp. 33.

[30] Stavros, H., Daniel J., A., Samuel, M., & Michael, S. (2008) 'OLTP through the looking glass, and what we found there', in Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. [Online]. 2013 ACM. pp. 981-992.

A Appendix: Bench-marking scripts

Listings 1 and 2 contain the scripts used to send the requests and store the results, which were stored in Excel sheets.

```
# vars
$VaultName = "******"
$UserName = "************"
$Password = "****************"

# The maximum number of objects that will be measured.
$Loops = 4500

# Create excel worksheet.
$excel = New-Object -ComObject Excel.Application
$excel.Visible = $true
$workbook = $excel.Workbooks.Add()
$worksheet = $workbook.Worksheets.Add()

# Initialize MFServer and log in to M-Files vault.
$mfserver = New-Object -ComObject MFilesAPI.MFilesServerApplication
$tzi = New-Object -ComObject MFilesAPI.TimeZoneInformation
$tzi.LoadWithCurrentTimeZone()
$mfserver.ConnectAdministrative( $tzi ) | out-null
$vaultGUID = $mfserver.GetVaults().GetVaultByName( $VaultName ).GUID
$testVault = $mfserver.LogInToVault($vaultGUID)


# Add column names to the worksheet.
$worksheet.Cells.Item(1,1) = "ObjectCount"
$worksheet.Cells.Item(1,2) = "Duration create"
$worksheet.Cells.Item(1,3) = "Duration checkin"

# Starts the upper loop, which tells the number of object we are now measuring.
for( ($m = 1); $m -lt $Loops; $m++)
{
    # Set starting point.
    $timecreate = 0
    $timecheckin = 0
    $timemetadatachange = 0

    # Inner loop that is responsible for sending the requests and storing the information.
    for( ($i = 0); $i -lt $m; $i++)
    {
            # Initialize required objects for the loops.
        $propertyName = New-Object -ComObject MFilesAPI.PropertyValue
        $propertyClass = New-Object -ComObject MFilesAPI.PropertyValue
        $propertyComment = New-Object -ComObject MFilesAPI.PropertyValue
        $typedValueName = New-Object -ComObject MFilesAPI.TypedValue
        $typedValueClass = New-Object -ComObject MFilesAPI.TypedValue
        $typedValueComment = New-Object -ComObject MFilesAPI.TypedValue
        $SourceObjectFiles = New-Object -ComObject MFilesAPI.SourceObjectFiles
        $AccessControlList = New-Object -ComObject MFilesAPI.AccessControlList
        $Properties = New-Object -ComObject MFilesAPI.PropertyValues
```

```
        $Properties1 = New-Object -ComObject MFilesAPI.PropertyValues

        # Add required values to create object.
        # name
        $fileName = "Document $(Get-Date -format 'u')"
        $typedValueName.SetValue(1, $fileName)
        $propertyName.PropertyDef = 0
        $propertyName.Value = $typedValueName

        # class
        $typedValueClass.SetValue(9, 1)
        $propertyClass.PropertyDef = 100
        $propertyClass.Value = $typedValueClass

        # Add both values to the properties list.
        $Properties.Add(0, $propertyName)
        $Properties.Add(1, $propertyClass)

        # Measure time spent in creating one object.
        $sw = [Diagnostics.Stopwatch]::StartNew()
        $document = $testVault.ObjectOperations.
                        CreateNewObject(0, $Properties, $SourceObjectFiles, $AccessControlList)
        $sw.Stop()
        $timecreate += $sw.Elapsed.TotalMilliseconds
        $sw.Reset()

        # Measure time spent in checking in one object.
        $sw.Start()
        $result1 = $testVault.ObjectOperations.CheckIn($document.ObjVer)
        $sw.Stop()
        $timecheckin += $sw.Elapsed.TotalMilliseconds
        $sw.Reset()

        # Change metadata information by adding one more property.
        $typedValueComment.SetValue(1, [string]$document.ObjVer.ID)
        $propertyComment.PropertyDef = 26
        $propertyComment.Value = $typedValueComment
        $Properties1.Add(0, $propertyComment)

        # Measure time spent in modifying the metadata information.
        $sw.Start()
        $result3 = $testVault.ObjectPropertyOperations.SetProperties($result1.ObjVer, $Properties1)
        $sw.Stop()
        $result3.Properties.ToJSON()
        $timemetadatachange += $sw.Elapsed.TotalMilliseconds
        $sw.Reset()
    }

    # Insert data to the excel worksheet.
    $worksheet.Cells.Item($m+1,1) = $m
    $worksheet.Cells.Item($m+1,2) = $timecreate
    $worksheet.Cells.Item($m+1,3) = $timecheckin
    $worksheet.Cells.Item($m+1,5) = $timemetadatachange

    # reset time for the next run.
    $sw.Reset()
}

# Save the workbook
```

```
$workbook.SaveAs("path\to\directory\name.xlsx")

# Close Excel
$excel.Quit()


# vars
$VaultName = "******"
$UserName = "***********"
$Password = "**************"

# The maximum number of objects that will be measured.
$Loops = 4500

# Create excel worksheet.
$excel = New-Object -ComObject Excel.Application
$excel.Visible = $true
$workbook = $excel.Workbooks.Add()
$worksheet = $workbook.Worksheets.Add()

# Initialize MFServer and log in to M-Files vault.
$mfserver = New-Object -ComObject MFilesAPI.MFilesServerApplication
$tzi = New-Object -ComObject MFilesAPI.TimeZoneInformation
$tzi.LoadWithCurrentTimeZone()
$mfserver.ConnectAdministrative( $tzi ) | out-null
$vaultGUID = $mfserver.GetVaults().GetVaultByName( $VaultName ).GUID
$testVault = $mfserver.LogInToVault($vaultGUID)

# Add column names.
$worksheet.Cells.Item(1,1) = "Lookup"
$worksheet.Cells.Item(1,2) = "range"

# Starts the upper loop, which tells the number of object we are now measuring.
for( ($m = 1); $m -lt $Loops; $m++)
{
    # Set starting point.
    $timelookup = 0
    $timerange = 0
    $timesearch = 0

    # Inner loop that is responsible for sending the requests and storing the information.
    for( ($i = 0); $i -lt $m; $i++)
    {
        # Initialize required objects.
        $typedValueName = New-Object -ComObject MFilesAPI.TypedValue
        $conditions = New-Object -ComObject MFilesAPI.SearchConditions
        $condition1 = New-Object -ComObject MFilesAPI.SearchCondition
        $condition2 = New-Object -ComObject MFilesAPI.SearchCondition
        $condition = New-Object -ComObject MFilesAPI.SearchCondition

        # Add search conditions.
        $condition1.ConditionType = 3
        $condition1.Expression.DataPropertyValuePropertyDef = 0
        $condition1.TypedValue.SetValue(1, "Document 2024-05-27 08:22:36Z")
        $condition2.ConditionType = 4
        $condition2.Expression.DataPropertyValuePropertyDef = 0
        $condition2.TypedValue.SetValue(1, "Document 2024-05-28 08:22:36Z")
        $conditions.Add(1, $condition1)
        $conditions.Add(2, $condition2)
        $condition.ConditionType = 1
```

```
        $condition.Expression.DataPropertyValuePropertyDef = 0
        $condition.TypedValue.SetValue(1, "Document 2024-05-27 08:22:36Z")

        # Measure look up search.
        $sw = [Diagnostics.Stopwatch]::StartNew()
        $result2 = $testVault.ObjectSearchOperations.SearchForObjectsByCondition($condition, $false)
        $sw.Stop()
        $timelookup += $sw.Elapsed.TotalMilliseconds
        $result2.Count
        $sw.Reset()

        # Measure range searchs.
        $sw.Start()
        $result3 = $testVault.ObjectSearchOperations
                        .SearchForObjectsByConditions($conditions, 2, $false)
        $sw.Stop()
        $result3.Count
        $timerange += $sw.Elapsed.TotalMilliseconds
        $sw.Reset()
    }

    # Add data to the excel worksheet.
    $worksheet.Cells.Item($m+1,1) = $m
    $worksheet.Cells.Item($m+1,2) = $timelookup
    $worksheet.Cells.Item($m+1,3) = $timerange

    # Reset the timer before the next loop.
    $sw.Reset()
}

# Save the workbook
$workbook.SaveAs("path\to\directory\name.xlsx")

# Close Excel
$excel.Quit()
```