Optimistic Concurrency Control and Multiversion Concurrency Control with MS SQL Server 2014

Seminar Database Systems
Master of Science in Engineering
Major Software and Systems
HSR Hochschule für Technik Rapperswil
www.hsr.ch/mse

Supervisor: Prof. Stefan Keller
Author: Michael Wolski

Zurich, 06. July 2014
Abstract

Keywords: In-Memory, Main-Memory, Database, Concurrency control, Optimistic concurrency control, Multiversion concurrency control, MS SQL Server 2014, SQL

Traditional databases were designed for hardware restrictions. Now hardware has more performance than ever. This evolution changes the architectural design of databases, so that they can perform on main memory. More and more database vendors changed the development from traditional databases to such called In-memory databases.

Performance is not only based on hardware, but also on algorithms and approaches. There are two approaches: pessimistic (lock-based) and optimistic. While optimistic can be extremely performant, because it is not using locks, but rules, the pessimistic approach has overhead of managing locks on data items and needs more resources. This paper is about the optimistic approach. Using Microsoft’s SQL Server 2014 gives some hands on with optimistic concurrency control.
# Table of contents

1 Overview.................................................................................................................. 5
  1.1 Goal ....................................................................................................................... 5
  1.2 Scope ...................................................................................................................... 6

2 Optimistic concurrency control .................................................................................. 7
  2.1 Introduction ............................................................................................................ 7
  2.2 Anomalies ............................................................................................................. 8
    2.2.1 Dirty read ........................................................................................................ 8
    2.2.2 Non-repeatable read ....................................................................................... 8
    2.2.3 Lost update/ Dirty Write .................................................................................. 9
    2.2.4 Write skew ...................................................................................................... 9
  2.3 ANSI SQL Isolation Levels ..................................................................................... 10
    2.3.1 Read uncommitted .............................................................. ................................ 10
    2.3.2 Read committed ............................................................................................. 11
    2.3.3 Repeatable read ............................................................................................. 11
    2.3.4 Serializable ................................................................................................. 11
    2.3.5 Snapshot isolation ......................................................................................... 11

3 Phases of a Transaction ............................................................................................. 11
  3.1 Memory-optimized tables ..................................................................................... 14
    3.1.1 Structure of a row .......................................................................................... 15
    3.1.2 Garbage Collector ......................................................................................... 15
    3.1.3 Indexes ........................................................................................................... 17
    3.1.4 Durability ....................................................................................................... 20
    3.1.5 Supported SQL isolation levels ................................................................. 21
    3.1.6 Further differences to disk-based tables .................................................. 21

4 Preparation .................................................................................................................. 22

5 Test cases ..................................................................................................................... 25
  5.1 Environment ......................................................................................................... 26
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.2</td>
<td>Snapshot isolation/Read committed</td>
<td>26</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Repeatable read</td>
<td>30</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Serializable</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Results</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>Conclusion and outlook</td>
<td>41</td>
</tr>
<tr>
<td>5</td>
<td>Bibliography</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>Figures</td>
<td>45</td>
</tr>
<tr>
<td>7</td>
<td>Tables</td>
<td>46</td>
</tr>
<tr>
<td>8</td>
<td>Listings</td>
<td>47</td>
</tr>
</tbody>
</table>
1 Overview

The history of database management systems (DBMS) began in 1960s, when hard disk speed and processors clock speed were the main components for a DBMS performance. Years after the trend went to main-memory or also called In-Memory databases. Data items are now accessible on the main-memory, which is the fastest way the processor can access and manage large data. The only way to get more performance out of a database is to optimize its structure, its behavior and its algorithms.

Modern hardware enables high transaction rates in DBMS and provides the possibility of concurrent transactions. The main quest is to find the best performing isolation level of executing concurrent transactions to allow respectively deny anomalies. These have influences on the database state.

One of the approaches is the optimistic concurrency control (OCC). The general purpose is to never lock data items during transactions to reach high transaction rates. The opposite is the pessimistic concurrency control (PCC), which only works with a lock manager. OCC makes sure that reading is always allowed and doesn't block other read operations. Reading operations are very usual use cases, i.e. search engines or online shops.

1.1 Goal

This paper gives an introduction of how optimistic concurrency control (OCC) and multiversion concurrency control (MVCC) works in theory and in practice using Microsoft SQL Server 2014. After reading this paper you are able to answer the following questions:

1. How do OCC and MVCC work and what are the best practices of using them?
2. What are the effects of changing the SQL isolation level on MVCC?
3. What are the advantages and disadvantages of MVCC?
4. How does Microsoft SQL Server 2014 support OCC, MVCC and the SQL isolation levels?

In the following we discuss the topics and show hands-on examples, which can be executed using the provided sources.

Chapter 2 is about the theory of optimistic concurrency control and multiversion concurrency control. There is also an introduction about database anomalies and SQL isolation levels, which are reliable on the behavior of a database..

Chapter 3 is the practice part. It is all about Microsoft SQL Server 2014 and some tests that will demonstrate the different anomalies with different SQL isolation levels. The examples used in this chapter can be downloaded and tested according to the system environment using in this paper.

Chapter 4 summarizes the main and important facts about OCC and gives a short comparison to alternative concurrency control approaches. It also includes a short outlook for the future.
1.2 Scope

The theory part (chapter 2) is generally written, so it corresponds to most database implementations. The practice part (chapter 3) and the provided sources are restricted to Microsoft SQL Server 2014. The test data and a testing application can be found on https://github.com/mwolski89/SQL-Isolation-Violation, which is still under development.
2 Optimistic concurrency control

This chapter is about optimistic concurrency control (OCC) based on the paper “High-Performance Concurrency Control Mechanisms for Main-Memory Databases” by “Per-Åke Larson1, Spyros Blanas2, Cristian Diaconu1, Craig Freedman1, Jignesh M. Patel2, Mike Zwilling”. It explains the approach of how to manage concurrent transaction executions and the necessary variables that affect the databases behavior.

Anomalies can lead transactions to change data item values in a wrong manner, so that the database state is inconsistent. Allowing or disallowing anomalies will affect the performance and stability of a database. These can be provoked with SQL isolation levels, which are explained in section 2.3.

During a transaction there are three steps before a commit can be done. These are necessary to reduce conflicts during concurrent actions on data items.

The last part is about multiversion concurrent control (MVCC). It is an implementation or extension to OCC. An Example shows how exactly versioning on data items are processed.

2.1 Introduction

OCC is a technique of concurrent transaction executions. The goal is to reach high transaction rates without locks.

Classic database management systems (DBMS) were designed for single transactions. These are using locks, to guarantee consistence and reduce failures and anomalies performing a transaction. This works similar to the “first-come-first-out”-paradigm. If a transaction performs a read operation it may be possible that other transactions will get locked. This case significantly reduces the systems transaction rate. Write operations do so respectively.

A read operation is not manipulating data, so there is no reason to lock this data. And this is what OCC does: Instead of allowing one read transaction on a data item, it allows concurrent transactions read data without locking. A detailed example can be found in section 2.5.

Write operations are more complex. If a transaction T1 updates a data item, the DBMS has to make sure that during T1 updates, other transaction did not modified this data item. If a change was made, T1 performs a rollback and must be recreated. Conflicts cost time and resources and reduce the performance. A detailed example can be found in section2.5.

OCC works best in cases where conflicts are rare, especially when reading data is a usual process. If concurrent UPDATE operations are processed, an optimistic approach is not recommended. The converse to OCC is pessimistic concurrency control (PCC). It uses locking mechanisms to reduce failures, if for example two transactions try to UPDATE a data item concurrently.
2.2 Anomalies

An Anomaly describes an inconsistence state or behavior of a database. In this section I will introduce four anomalies defined in [1].

2.2.1 Dirty read

Dirty read is an anomaly, when a transaction T1 changes a dataset, but did not commit and transaction T2 reads the same dataset and read the changed values.

In [1] Dirty Read is described as follows: "Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed."

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Employee Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE Employee</td>
<td></td>
<td>Smith</td>
</tr>
<tr>
<td>SET Lastname = 'Carter'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHERE EmployeeId = 1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SELECT Lastname</td>
<td>Carter</td>
</tr>
<tr>
<td></td>
<td>FROM Employee</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE EmployeeId = 1234</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rollback</td>
<td>Smith</td>
</tr>
</tbody>
</table>

Figure 1: Dirty read example

2.2.2 Non-repeatable read

Non-repeatable read is an anomaly, when the result set of the same selection returns different values.

In [1] Non-repeatable read is described as follows: "Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted."

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Employee Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT Lastname</td>
<td></td>
<td>Smith</td>
</tr>
<tr>
<td>FROM Employee</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WHERE EmployeeId = 1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UPDATE Employee</td>
<td>Carter</td>
</tr>
<tr>
<td></td>
<td>SET Lastname = 'Carter'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE EmployeeId = 1234</td>
<td>commit</td>
</tr>
<tr>
<td></td>
<td>SELECT Lastname</td>
<td>Carter</td>
</tr>
<tr>
<td></td>
<td>FROM Employee</td>
<td></td>
</tr>
<tr>
<td></td>
<td>WHERE EmployeeId = 1234</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Non-repeatable read example
2.2.3 Lost update/ Dirty Write

Lost update describes an anomaly when a transaction makes changes to a data item, but instead another transactions changes will be used. Example:

Transaction T1 updates a data item. At the same time transaction T2 makes also changes to that data item and commits the changes. After T2 commit was successfully, T1 commits its changes. The changes of T2 got overridden.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Employee Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Smith</td>
</tr>
<tr>
<td>UPDATE Employee</td>
<td>SET Lastname = 'Carter'</td>
<td>WHERE EmployeeId = 1234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Carter</td>
</tr>
<tr>
<td>UPDATE Employee</td>
<td>SET Lastname = 'Wayne'</td>
<td>WHERE EmployeeId = 1234</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>Wayne</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>Carter</td>
</tr>
</tbody>
</table>

**Figure 3: Lost update example**

2.2.4 Write skew

Write skew is an anomaly in concurrent environments (such as snapshot isolation and serializable) and occurs when restrictions for data items got broken. Example:

In the database attribute A1 and attribute A2 have a restriction to each other. Transactions T1 and T2 are reading a data item. After the reading process both transactions got a local copy. Now transaction T1 changes A1. T2 changes A2. Both changes satisfy the restriction on their local copies. Now both T1 and T2 commit their changes, but the restrictions are not checked on the database. The result is a data item with broken restrictions, because T1 only changes A2 and T2 only changes A1.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>Salden für Konto1, Konto2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Konto1: 100, Konto2: 100</td>
</tr>
<tr>
<td>SELECT Konto1, Konto2</td>
<td>FROM Konto</td>
<td>WHERE KtoNr = 1234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Konto1: 100, Konto2: 100</td>
</tr>
<tr>
<td>SELECT Konto1, Konto2</td>
<td>FROM Konto</td>
<td>WHERE KtoNr = 1234</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Konto1: 100, Konto2: 100</td>
</tr>
<tr>
<td>UPDATE Konto</td>
<td>SET KtoNr=100 -120</td>
<td>WHERE KtoNr = 1234</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>Konto1: -20, Konto2: 100</td>
</tr>
<tr>
<td>UPDATE Konto</td>
<td>SET KtoNr=100 -120</td>
<td>WHERE KtoNr = 1234</td>
</tr>
<tr>
<td></td>
<td>commit</td>
<td>Konto1: 100, Konto2: -20</td>
</tr>
</tbody>
</table>

**Figure 4: Write skew example**
2.3 ANSI SQL Isolation Levels

In some cases, especially in concurrent environments, databases can have different phenomena. This can be controlled by the use of isolation levels. Lower levels improve transaction rates in concurrent cases but are more error prone (i.e., incorrect data items), while the highest level can guarantee “perfect serializable” transactions. Latter reduces failure if for instance one transaction changes data without performing a commit and another transaction can read the uncommitted changes. Managing the behavior of transactions or databases can be very confusing.

The ANSI/ISO SQL-92 specification by the American National of Standards Institute defines four isolation levels with their anomalies respectively anomalies (see section 2.2).

<table>
<thead>
<tr>
<th>Isolation Level/Anomalies</th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
<th>Lost update</th>
<th>Write skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read uncommitted</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
<td>-</td>
</tr>
<tr>
<td>Read committed</td>
<td>impossible</td>
<td>possible</td>
<td>possible</td>
<td>-</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>impossible</td>
<td>impossible</td>
<td>possible</td>
<td>-</td>
</tr>
<tr>
<td>Snapshot isolation</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible</td>
<td>possible</td>
</tr>
<tr>
<td>Serializable</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible</td>
</tr>
</tbody>
</table>

Table 1: Isolation Levels and their anomalies

Table 1 shows which anomalies are possible on selected isolation levels. Section 3.3 is all about the proof of these behaviors in MS SQL Server 2014.

In the following sections I will explain the effects of the isolation levels and why anomalies are/are not allowed.

2.3.1 Read uncommitted

Read uncommitted is an isolation level, which allows all anomalies. Uncommitted transactions are visible to other users. So dirty reads are always possible, because changes, which are not committed, are always set in the database. This can also have the effect when querying many times the same data set (non-repeatable read). Of course when another transaction changes a value on a data item, changes from other transactions get lost (Lost update).
2.3.2 Read committed

Read committed is an isolation level, which allows all anomalies except dirty reads. This has the advantage that changes of a transaction are not visible to other transactions until a commit is performed. But non-repeatable reads and lost updates are also allowed.

2.3.3 Repeatable read

Repeatable read is an isolation level, which only allows lost updates. During a selection in a transaction other transactions are not allowed to change the selected data items. A lock on the selected data items makes sure that no dirty reads and non-repeatable reads are allowed. The data items get unlocked after the transaction ends.

2.3.4 Serializable

Serializable is the highest isolation level. The idea is that transactions will be performed sequentially, so that transactions have no effects on each other. This makes sure that no dirty writes, non-repeatable reads and no lost updates are possible.

2.3.5 Snapshot isolation

Snapshot isolation (SI) is defined in the ANSI/ISO SQL-92 specification as a “type of multiversion concurrency control” [1]. SI never blocks read operations. Write operations will be performed on a local snapshot, which is isolated to other transactions, so changes will not be visible to other transactions until a commit. Before a commit is performed, SI validates if the committing data was not changed on the database since the last snapshot. SI uses timestamps to see if a version of a data item is the latest (more details in section 2.5). If there is a higher start-timestamp of one of the changed data items on the database, the validation fails and the commit aborts. The transaction has to rollback and repeat the transaction. This leads to a problem called write-write-conflict (see 2.5). Otherwise, when the validation passes, the commit can be executed. This behavior has the advantages that most anomalies are not possible.

2.4 Phases of a Transaction

For a better understanding of OCC, we should take a look on how transactions are performed. Usually there are three types or phases of a transaction: read, validation and write. Each of them with different tasks and processing time. The implementation of these phases has impact on the performance of a database and its consistency. The validation can be set with one of the SQL isolation levels. In the following I will introduce into the specific phases and the main tasks of the phases.

2.4.1 Read

During the reading phase the database collects a set of data items based on given criteria and serves them. This is a simple task. During this phase conflicts are not possible, even
when concurrent transactions are reading. OCC works best when performing reading operations, so this should be the main used transaction phase.

2.4.2 Validation

The validation phase gets started, when a transaction performs a commit. The validation phase is based on the chosen SQL isolation level. Let's assume transaction T1 use read uncommitted as SQL isolation level. This means changes are made without a commit. In this case the validation process would do nothing, because every anomaly is allowed. A higher isolation level would make the validation more complex.

Let's assume T1 uses snapshot isolation. This means we have a snapshot of the data base which other transaction can't see. Now we change a data item and want to commit. The validation process now compares if the data items are equally to the data items in the database before the transaction updates. If the transaction updates a large set of data, this process would take a lot of time. Only if all data are equal, the validation is true and a commit can be performed. If not, T1 must do a rollback, T1 has to be recreated and the update process must be repeated.

2.4.3 Write

After the validation is successfully done, the transaction starts to update the data items. Especially in MVCC, a write creates a new version of the updated data item and insert it into the database. The new added data item is marked as the current one and the last version will be marked as deprecated (see section 2.5). This makes sure that read operations will not be blocked, when a write operation is performed.

2.5 Multiversion concurrency control

MVCC is an implementation of OCC. As the name suggests, versioning is used to manage data items. Each write operation creates a new version of the edited data item, instead of overwriting it. The old/original version is available as long as there is a transaction that uses this version. For efficiency memory usage a garbage collector should delete versions, which are not used.

Read operations always get the latest/current version of a data item. So if changes are made, readers will see them directly after a write operation was successfully performed. The following example shows how MVCC basically works in SQL Server 2014:

The SQL Server 2014 introduces a new format to store data, called memory-optimized tables. This format differs from traditional disk-based tables, which will be introduced in section 3.1. The data structure has the attributes “Begin Timestamp” (BTS) and “End Timestamp” (ETS)
Optimistic Concurrency Control and Multiversion Concurrency Control with MS SQL Server 2014

Figure 5: Row in multiversion concurrency control

(see Figure 5). BTS is always the creation timestamp of this version. If this data item is the latest then the ETS value is unlimited (∞). If a new version will be created the ETS of the last version changes to the BTS of the new version: ETS₀ == BTS₁.

Reading operations will not create new versions of data items, so there is nothing to do in context of versioning. But in case that there is a write operation, a new version will be created, and after a commit a validation must be done. A practical scenario for validation failure is the write-write-conflict (see Figure 6). When two concurrent transactions try to commit their changes, one will fail. As well as OCC, MVCC has no solution for this case. Assume there are two transactions T₁ and T₂. T₁ changes the Lastname of the employee with an EmployeID = 1234 to “Wayne” but did not commit. T₂ changes the Lastname of the same employee to “Carter” and commits now. During the commit the validation process examines there was a change on this data item in the database, in this case no because T₁ does not commit. So the commit is completed and T₂ finishes. Then T₁ commits, but does not pass the validation process, because this time there was a change. The result is, T₂ commits successfully, while T₁ gets an error. Now T₁ must do a rollback, recreate itself and must try to recommit the changes.

Write operations can generate a huge amount of overhead, especially if there are conflicts. Conflicts are not affecting concurrent reading transactions so a failure does block other transactions.

Figure 6: Write-write-conflict
3 MVCC in Microsoft SQL Server 2014

"Traditional RDBMS architecture was designed when memory resources were expensive, and was optimized for disk I/O" [2]. As Hardware limitations and prices got lower, especially RAM (Random Access Memory), new approaches are possible. In-Memory databases were designed and data is outsourced on main memory. Main memory is quite faster than a hard disk (even faster than expensive SSDs (Solid State Disk) and database vendors start to develop In-memory database products i.e. Oracle [3], IBM [4], SAP [5] etc. On April 2014 Microsoft released the SQL Server 2014. The improvements were mainly focused on In-memory capabilities, Microsoft calls it In-Memory OLTP, realized with the “Hekaton” engine [6]. The SQL Server 2014 got a new type of a table, called memory-optimized table, which stores data in an optimized structure in main memory to gain faster transaction executions. Microsoft decides to implement memory-optimized tables using optimistic multiversion concurrency control (MVCC) [7]. As already described in section 2.5, MVCC has the advantage of lock-free transactions. Unlike disk-based tables, a memory-optimized table creates a new version of a data item instead of overwriting it. This was one reason to develop a new type of table. For our goal to learn how SQL isolation levels affect MVCC, the SQL Server 2014 is a good starting point. In section 3.3 the SQL isolation levels will be violated, to verify if a database acts as expected. This chapter is all about MS SQL Server 2014. In the part I will give a short introduction to SQL Server 2014, especially the new memory-optimized table. The second part is about the test cases, which visualize what effects SQL isolation levels have on a database using a MVCC table.

3.1 Memory-optimized tables

From the technology perspective In-memory databases use the main memory of a computer to store and access data. In comparison to hard disks, main memory is often limited (less memory space) and data has to be stored and managed in a resource-friendly manner. Another big disadvantage of storing data in main memory is availability. When main memory is out of power, all data will be lost. So there has to be a solution to backup or rather restore data in main memory. With these bottlenecks in mind, Microsoft introduces memory-optimized tables in Microsoft SQL Server 2014. Memory-optimized tables implements optimistic multiversion concurrency control (MVCC). Data is stored in an efficient manner in main memory. Section 3.1.1 shows how is the structure of the data in terms of memory-optimized tables. Because of memory capacity and fast operations Microsoft changes some processes like logging and reduced it to a minimum of overhead. Section 3.1.4 shows how backups for memory-optimized tables will be generated.

MVCC creates a new version of a data item in when an UPDATE operation is executed. Many versions need more memory, so it will lead to a memory space problem if old versions are
hold. To manage that Microsoft implements a garbage collector to release memory if possible. The garbage collector is explained in section 3.1.2.

Microsoft introduced hash and nonclustered indexes for memory-optimized tables. These are designed for every day cases and differ from disk-based table indexes. Both index structures will be introduced in section 3.1.3.

Memory-optimized tables have limitation in using SQL isolation levels. These will be discussed in section 3.1.5. Further differences in comparison to disk-based tables, which for example could affect a migration project are explained in section 3.1.6.

### 3.1.1 Structure of a row

Microsoft designed a new structure of rows for memory-optimized tables. They are designed with the concept of MVCC. A row is split in two main elements: row header and payload. Figure 7 shows a detailed version of Figure 5.

![Figure 7: The structure of a row in a memory-optimized table, Source: [8], “Rows”](image)

The payload has all the columns data, defined in the database (by the user). The main part which makes the structure special for MVCC is the row header. The row header has the following attributes:

- **Begin Ts**: timestamp of the moment, when the row was added to the database.
- **End Ts**: timestamp of the moment when the row has been deleted or updated. If this row version is the latest this value is set to ‘infinite’.
- **Stmlid**: statement ID of the create statement of the row, which is set by the database.
- **IdxLinkCount**: the number of references from the index table.
- **Last two attributes**: are pointers between the index table and the row.

### 3.1.2 Garbage Collector

MVCC creates one or more versions of a data item. This approach store many different versions. Especially when (limited) main memory is used, there has to be a solution to
manage old versions. If memory reaches its limits, the whole system may crash and be unstable.

With that in mind Microsoft implements the garbage collector as a solution. It is the most important component when using MVCC. Different from garbage collection (GC) in programming languages, it is used in terms of MVCC to remove old versions, which are no longer needed by any transaction. Microsoft describes the design principles of GC in Hekaton as follows:

- **Hekaton GC is non-blocking.** Garbage collection runs concurrently with the regular transaction workload, and never stalls processing of any active transaction.
- **The GC subsystem is cooperative.** Worker threads running the transaction workload can remove garbage when they encounter it. This can save processing time as garbage is removed proactively whenever it is “in the way” of a scan.
- **Processing is incremental.** Garbage collection may easily be throttled and can be started and stopped to avoid consuming excessive CPU resources.
- **Garbage collection is parallelizable and scalable.** Multiple threads can work in parallel on various phases of garbage collection and in isolation with little to no cross-thread synchronization.

(Source: [9])

If old versions of a data item is no longer needed, it is called “garbage”. This means that it can be removed and everything still works fine. There are three scenarios of how version can change to be garbage:

- By an explicit DELETE or UPDATE operation that was committed.
- The version cannot be read or is not needed by any transaction.
- A transaction do a rollback after it creates a new version.

Like described in section 3.1.1, the value end timestamp indicate that this version is no longer visible to other transactions, because there is a newer version. “Any version whose end timestamp is less than the current oldest active transaction in the system is not visible to any transaction and can be safely discarded” (Source: [9]).

The GC periodically starts a thread to remove data. In that case there are three processes:

- **Transaction mapping processing:** First, the thread looks for the begin timestamp for the oldest active transaction. The garbage collector will be notified to then collect all closed transactions ordered by the end timestamp and process the garbage version collecting.
- **Garbage version collecting:** The thread now collects all deleted and updated data items by every transaction and stores them in the garbage collection.
- **Unlinking:** The GC uses two methods to unlink garbage version: cooperative mechanism and a parallelized background processing process. The last one happens during the thread in terms the GC starts it. The cooperative mechanism is integrated i.e. indexing process, so when indexing encounters a garbage version, it simply unlinks it. This works well especially for “hot” areas, because the indexing process knows in point of time that this is a garbage version. But “cold” areas are not in scope. In that case there is the parallelized background process is used process
through the index tables to unlink the pointers. If unlinking is complete, these garbage versions can be removed.

After these steps the garbage collector frees memory. More about the garbage collector can be found in [9] in chapter 8.

### 3.1.3 Indexes

Indexes are used to tune query performance. It helps to find an object or a range of objects in short time. Microsoft implements two new indexes for memory-optimized tables in SQL Server 2014: Hash indexes and nonclustered indexes. Both can be combined in one table and bring improvements in special cases. Memory-optimized tables have limitation in terms of indexes. When defining an indexing strategy, the following restrictions should be in mind:

- **At most 8 indexes** – you cannot specify more than 8 indexes on a single memory-optimized table.
- **BIN2 collations** – when using n(var)char columns in an index key, the columns must have a _BIN2 collation. Using BIN2 allows very fast lookup, as this can be based simply on binary comparison. However, you need to consider the implications of using a BIN2 collation, such as case and accent sensitivity. For more details see the Books Online topic on Collations and Code Pages.
- **NOT NULL columns in the index key** – memory-optimized indexes do not support nullable columns; all columns in the index key must be declared as NOT NULL.

(Source: [10])

Memory-optimized tables must have minimum one index (usually on the primary key) and other than in disk-based tables, they can only be created during a CREATE TABLE operation. To create a new index key for the table, the table has to be dropped and recreated. A second difference to disk-based tables is that indexes are not durable or logged in the transaction log record. Every time the database restarts, the database creates new index keys.

In the following subsections I will describe how the indexes are working.

#### 3.1.3.1 Hash index

A hash index is a newly introduced index type. It is stored in a hash table and performs best in cases where point lookups respectively equality queries are performed. A hash table saves all hash index keys with a pointer to the row of the table and places them in random order in a so called bucket. In fact a hash table is an array and a bucket is an element in this array.
CREATE TABLE [Sales].[SalesOrderHeader_inmem](
    [SalesOrderID] uniqueidentifier NOT NULL
    PRIMARY KEY
    NONCLUSTERED HASH WITH (BUCKET_COUNT=1000000)
    DEFAULT (NEWID()),
    [RevisionNumber] [tinyint] NOT NULL CONSTRAINT
    [IMDF_SalesOrderHeader_RevisionNumber] DEFAULT ((0)),
    [OrderDate] [datetime2] NOT NULL
INDEX ix_OrderDate HASH WITH (BUCKET_COUNT=1000000)
) WITH (MEMORY_OPTIMIZED=ON)

Listing 1: CREATE TABLE operation with a hash index, Source: [8]

shows an example of how to create a table with a hash index (in the second last line). When creating a hash index the variable BUCKET_COUNT has to be set. Microsoft recommends to use a value which is one or two times of the number of unique index key values. A to low value has negative impact on the performance, but a to high value increases memory-consumption and slows down full index scans (see [10]).

Figure 8: A hash index with a single row, Source: [8]

Figure 8 is an example of how a hash index key will be stored in a hash table. Let's assume that the hash function generates an index key based on the length of the name value. This is just to make the example easier to understand. In fact it places a bucket in random order. Because the word “Jane” has four digits, a bucket with the key value (in this case four) and the name “Jane” is placed in position four of the hash table in our example. The attribute “index ptr” is the index pointer. The index pointer value references to another row, if there is more than one row for the same key value. The other attributes in the example are derived from the example shown in section 2.5. It is just important to understand that in this case a bucket is added to the hash table in position four.
It is possible that the hash function generates index key for a row, which already exist in the hash table. In this case the new bucket is added to the hash table into the already written bucket and references to the old/last bucket over the index pointer attribute. Figure 9 demonstrates such a case. A new row with the value “Greg” (it also has 4 digits) for the name is added to the database. The hash function generates the same value for the index key.

![Figure 9: A hash index with two rows, Source: [8]](image)

The Greg row is now on the top of the bucket and references to the Jane row. If more rows will be added it generates a chain of index pointers, so every row is referenced with the same index key.

### 3.1.3.2 Nonclustered indexes

A hash index has the disadvantage that rage queries cannot profit from the index. So it will start to make an expensive full table scan. Another disadvantage occurs when choosing a wrong value for the BUCKET_COUNT. To have an alternative to hash indexes Microsoft implements nonclustered indexes.

These indexes don’t use the BUCKET_COUNT attribute and are best in cases when range queries are performed.

```sql
CREATE TABLE [Sales].[SalesOrderHeader_inmem](
    [SalesOrderID] uniqueidentifier NOT NULL
    PRIMARY KEY
    NONCLUSTERED HASH WITH (BUCKET_COUNT=1000000)
    DEFAULT (NEWID()),
    [RevisionNumber] [tinyint] NOT NULL CONSTRAINT [IMDF_SalesOrderHeader_RevisionNumber] DEFAULT ((0)),
    [OrderDate] [datetime2] NOT NULL
    INDEX ix_OrderDate NONCLUSTERED
) WITH (MEMORY_OPTIMIZED=ON)
```

Listing 2: CREATE TABLE operation with a nonclustered index, Source: [7]
Listing 2 shows how a nonclustered index can be created. It must be created during CREATE TABLE operation. To add an index to an existing memory-optimized table the table needs to be recreated with the second last line in Listing 2.

```sql
SELECT * FROM Sales.SalesOrderHeader_inmem
WHERE OrderDate > @Date
```

**Listing 3: SELECT operation with range, Source: [10]**

Executing a query like in Listing 3 with a range will give an ordered ascending list. If the query would have an “ORDER BY OrderDate” instead of the WHERE clause, there would be no further processing because the indexes are already sorted in ascending order. When creating the index (like in Listing 2) the order can be set (default is ASC=ascending). The big disadvantage in nonclustered indexes is, that it is not possible to get a descending list when ascending order is set. In fact if such a query would be executed the index ix_OrderDate wouldn’t be used.

### 3.1.4 Durability

“In-memory OLTP provides full durability for memory-optimized tables” [11]. A Memory-optimized table have two key components to guarantee full durability, based on disk-based tables, but with less processing and resource-efficient usage:

- **Transaction Logging** captures all changes to a transaction log record. Unlike disk-based tables, memory-optimized tables are only capturing changes after a transaction begins a commit operation. Rollback operations are not captured. In fact a log record includes all inserts and deletes of row versions related to a table. Disk-based tables implement write-ahead logging (WAL), which means that transactions will be captured in the log record before they are successfully committed in the database. So WAL would for instance log dirty reads. Memory-optimized tables makes changes only when transactions are committed. So WAL processing is not in scope. Changes to indexes in memory-optimized tables will not be logged, because they are not persistent. Indexes will always be re-generated (i.e. when database restarts).

Multiple transactions will be captured in large groups of transaction log records (currently up to 24 kilobytes). It is recommended to use less, but larger records, to reduce the overhead of managing log-headers. In comparison to disk-based tables, memory-optimized tables generate less log data and fewer write operations. Memory-optimizes tables gain processing time and effective resource usage (see [11]).

- **A Checkpoint** is a physical file on the hard disk in the SQL Server 2014 environment, generated by the data and delta file based on the transaction log records. Rows from one or more memory-optimized tables generated by INSERT and UPDATE operations will be stored in the data file. DELETE
operations will store the deleted rows in the delta file. It is possible that there are more than one data and delta file, but there is always a delta file for every data file. Both are allocated in the main memory. In case of a crash, the transaction log record is located on the hard disk. The SQL Server 2014 has a limitation of 8192 pairs (data + delta = 1 pair).

A checkpoint file can be created automatically (starting a background process when the log record reaches 1 Gigabyte after the last checkpoint creating) or manually. If a checkpoint file already exists, it will be appended. The data and delta files will be flushed to the hard disk. The checkpoint will be constructed, based on the log records and the data and delta files. Only existing rows will be added to the checkpoint. If the SQL Server crashes, it recovers the last state of the table using the checkpoint file (see [11]).

Transaction Logging and checkpoints implement solutions to backup memory-optimized tables, and offer a great way to recover them automatically.

### 3.1.5 Supported SQL isolation levels

Memory-optimized tables only supports three isolation level:

- SNPASHOT ISOLATION
- REPEATABLE READ
- SERIALIZABLE

Other than disk-based tables, memory-optimized tables use snapshots for these three isolation levels, so that every transaction operates in its own context. READ_COMMITED in memory-optimized tables is only supported when the option “autocommit transactions” are enabled. This option commits transactions in the background, when a transaction performs an operation (even when there is no COMMIT statement).

### 3.1.6 Further differences to disk-based tables

Memory-optimized tables differs from disk-based tables in many ways. This chapter is about all major differences in SQL Server 2014.

#### 3.1.6.1 Supported data types

A memory-optimized table supports only the following data types:

- Bit
- All integer types: tinyint, smallint, int, bigint
- All money types: money, smallmoney
- All floating types: float, real
- date/time: datetime, smalldatetime, datetime2, date, time
- numerical and decimal types
- All non-LOB string types: `char(n)`, `varchar(n)`, `nchar(n)`, `nvarchar(n)`, `usname`
- Non-LOB binary types: `binary(n)`, `varbinary(n)`
- `Uniqueidentifier`

(Source: [8], “Creating Databases”)

### 3.1.6.2 Length limitation

In memory-optimized tables have limitations in the size of rows. The maximum size of a row is limited to 8060 bytes. Means that two columns with an attribute of `varchar(5000)` are not valid. The limitation is checked during table creation. (Source: [8], “Creating Databases”)

### 3.1.6.3 Further differences

During the creation of a memory-optimized tables there are a few more restrictions in comparison to disk-based tables:

- No DML triggers
- No FOREIGN KEY or CHECK constraints
- No UNIQUE indexes other than for the PRIMARY KEY
- A maximum of 8 indexes, including the index supporting the PRIMARY KEY

(Source: [8], “Creating Databases”)

Another change was made to indexes. It is not possible to add an index to an active memory-optimized table. Indexes must be created during table creation. If an index should be added to an existing memory-optimized table, the table has to be recreated with the index. (Source: [8], “Creating Databases”)

### 3.2 Preparation

There is a testing application developed in Java that will be further developed and can be downloaded from [https://github.com/mwolski89/SQL-Isolation-Violation](https://github.com/mwolski89/SQL-Isolation-Violation). There has to be an SQL Server 2014 instance installed, before the test can be started!

The application provides a setup function which will generate a database with a table. The size of the test data is not necessary, because we only will perform two concurrent transactions. Further instructions can be found on the Github page.

The test needs two tables: `persons` (Listing 4) and `accounts` (Listing 5).
USE [ilv]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[persons]
(
    [EmployeeID] [int] NOT NULL,
    [Firstname] [nvarchar](45) COLLATE Latin1_General_100_BIN2 NOT NULL,
    [Lastname] [nvarchar](45) COLLATE Latin1_General_100_BIN2 NOT NULL,

INDEX [IDXFirstname] NONCLUSTERED HASH
(    [Firstname]
)WITH ( BUCKET_COUNT = 1024),
INDEX [IDXLastname] NONCLUSTERED
(    [Lastname] ASC
),
CONSTRAINT [PKEmployeeID] PRIMARY KEY NONCLUSTERED
(    [EmployeeID] ASC
)WITH ( MEMORY_OPTIMIZED = ON, DURABILITY = SCHEMA_AND_DATA )
GO

Listing 4: person table
USE [ilv]
GO
SET ANSI_NULLS ON 
GO
SET QUOTED_IDENTIFIER ON 
GO
CREATE TABLE [dbo].[accounts]
(    [AccountID] [int] NOT NULL,    [Amount1] [int] NOT NULL,    [Amount2] [int] NOT NULL,)
INDEX [IDX_amount2] NONCLUSTERED
(    [Amount2] ASC ),
CONSTRAINT [PK_AccountID] PRIMARY KEY NONCLUSTERED
(    [AccountID] ASC )
)WITH ( MEMORY_OPTIMIZED = ON , DURABILITY = SCHEMA_AND_DATA )
GO

Listing 5: Table accounts

For the test cases the Java tool from GitHub is used to visualize the queries and results. The next sections are about the processes and behaviors of the database using all discussed isolation levels in section 2.3 and validating, if anomalies from section 2.2 acting as expecting.

There is a limitation when testing write skew anomaly: To develop a constraint environment on two attributes as described in section 2.2.4 there would be a so called CHECK Constraint in SQL Server which would allow to set a restriction on attributes, like amount1 + amount2 >= 0. But a memory-optimized table does not support CHECK constraints, instead Microsoft recommends to use stored procedures [12]. In this case we would need two stored procedures to validate the amounts attributes (see Listing 6). This results as Microsoft technically does not make it possible to have write skews. The missing feature of using CHECK constraints makes it impossible to have a write skew, means a broken restriction on data.
USE [ilv]
GO
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO

CREATE PROCEDURE [dbo].[Check_Amount_Is_Positive_Amount1]
    @AccountID int, @Amount1 int
AS
BEGIN
    BEGIN TRANSACTION
    BEGIN TRY
        DECLARE @Amount2 INT
        SELECT @Amount2 = Amount2 FROM dbo.accounts WITH(SNAPSHOT) WHERE AccountID = @AccountID;
        IF NOT (@Amount1 + @Amount2 >= 0)
            THROW 50547, N'The sum of the amounts is smaller than zero.', 10
        UPDATE ilv dbo.accounts WITH(SNAPSHOT) SET Amount1 = @Amount1, Amount2 = @Amount2 WHERE AccountID = @AccountID
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;
            THROW;
    END CATCH
END

CREATE PROCEDURE [dbo].[Check_Amount_Is_Positive_Amount2]
    @AccountID int, @Amount2 int
AS
BEGIN
    BEGIN TRANSACTION
    BEGIN TRY
        DECLARE @Amount1 INT
        SELECT @Amount1 = Amount1 FROM dbo.accounts WITH(SNAPSHOT) WHERE AccountID = @AccountID;
        IF NOT (@Amount1 + @Amount2 >= 0)
            THROW 50547, N'The sum of all amounts is smaller than zero.', 10
        UPDATE ilv dbo.accounts WITH(SNAPSHOT) SET Amount1 = @Amount1, Amount2 = @Amount2 WHERE AccountID = @AccountID
    END TRY
    BEGIN CATCH
        IF @@TRANCOUNT > 0
            ROLLBACK TRANSACTION;
            THROW;
    END CATCH
END

Listing 6: Stored procedure to validate amount1 and amount2

3.3 Test cases

The test cases are categorized by the SQL isolation levels introduced in section 2.3, and get tested against anomalies defined in section 2.2. Currently there are restrictions in memory-
optimized tables, so it is not possible to test all SQL isolation levels (see section 3.1.5, Fehler! Verweisquelle konnte nicht gefunden werden.).

In Fehler! Verweisquelle konnte nicht gefunden werden., there are all listed test cases with anomalies which are expected to occur using the specified isolation level. Write skew can only be tested on snapshot isolation and serializable level, because it is a concurrency anomaly. The test visualizes how the behavior of the database when using an SQL isolation level is.

3.3.1 Environment

The test cases are performed in Microsoft SQL Server 2014 Enterprise Edition (64-bit) under Windows 8.1 (64-bit). The supporting tool for querying the database is the Java tool from GitHub. For own purpose it is also possible to use MS SQL Server 2014 Management Studio, which is included in the MS SQL Server Suite.

3.3.2 Snapshot isolation/Read committed

Snapshot isolation was discussed in section 2.3.52.3.3 and is supported by MS SQL Server 2014. It is only available in combination with READ_COMMITTED_SNAPSHOT for versioned rows [13]. Microsoft sets snapshot isolation as the lowest isolation level which in fact acts like a read committed isolation level, but makes a snapshot of the current database during a transaction. Other transaction cannot make changes on that snapshot. It is possible to set explicit the read committed isolation level above of the BEGIN TRANSACTION query: SET TRASNACITION ISOLATION LEVEL READ COMMITTED, but is not necessary, because transactions without an explicit transaction level is default read committed. In fact there is no snapshot isolation like defined in section 2.3.5.

3.3.2.1 Dirty read

Listing 7 shows the steps to provoke a dirty read. As the result shows, uncommitted transactions have no effect on other transactions. The change will take effect after a commit.
---DIRTY READ

Transaction t2:
SELECT [EmployeeID], [Firstname], [Lastname] FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t1:
BEGIN TRANSACTION t1

Transaction t1:
SELECT [EmployeeID], [Firstname], [Lastname] FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Carter

Transaction t2:
SELECT [EmployeeID], [Firstname], [Lastname] FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t1:
ROLLBACK TRANSACTION t1;

Transaction t2:
SELECT [EmployeeID], [Firstname], [Lastname] FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith

Listing 7: Dirty read in snapshot isolation

3.3.2.2 Non-repeatable read
In Listing 8 are the steps to have a non-repeatable read anomaly as described in section 2.2.2. The result shows that the anomaly is possible as the same query shows different results, when another transaction updates an attribute.
---NON-REPEATABLE_READ

Transaction t1:
SELECT [EmployeeID] 
, [Firstname] 
, [Lastname] 
FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1 Firstname: John  Lastname: Smith
EmployeeID: 2 Firstname: Jerry  Lastname: Springer

Listing 8: Non-repeatable read in snapshot isolation

3.3.2.3 Lost update
As described in section 2.2.3 a lost update appears when a write operation of a transaction gets lost. The steps to provoke a lost update can be seen in Listing 9. The result shows that, a lost update is not possible. But this results in the fact that when a data item was changed by transaction t1, transaction t2 is not able to update the same attribute. So the UPDATE step of t2 remains in a write-write-conflict, which forces t2 to make a ROLLBACK and prints an error message.
************LOST UPDATE
########################################################################
Transaction t1:
SELECT [EmployeeID]
      ,[Firstname]
      ,[Lastname]
FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith
EmployeeID: 2  Firstname: Jerry  Lastname: Springer

########################################################################
Transaction t1:
BEGIN TRANSACTION t1
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Wayne'
WHERE [EmployeeID] = '1';

########################################################################
Transaction t2:
BEGIN TRANSACTION t2
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Carter'
WHERE [EmployeeID] = '1';

com.microsoft.sqlserver.jdbc.SQLServerException: The current transaction attempted to update a record that has been updated since this transaction started. The transaction was aborted.

########################################################################
Transaction t2:
COMMIT TRANSACTION t2;

########################################################################
Transaction t1:
COMMIT TRANSACTION t1;

########################################################################
Transaction t2:
SELECT [EmployeeID]
      ,[Firstname]
      ,[Lastname]
FROM [ilv].[dbo].[persons] WITH(SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Wayne
EmployeeID: 2  Firstname: Jerry  Lastname: Springer

Listing 9: Lost update in snapshot isolation

3.3.2.4 Write skew

As mentioned in section 2.3.5, write skew is an anomaly, where restrictions got broken. This is only possible in snapshot isolation.

In Listing 10, you can see the steps to provoke a write skew anomaly. The last select statement shows that, a write skew anomaly in Snapshot Isolation is not allowed, even possible.
Listing 10: Write skew in snapshot isolation

3.3.3 Repeated read

Repeatable read was discussed in section 2.3.3 and is supported by MS SQL Server 2014.

3.3.3.1 Dirty read

Performing the steps in Listing 11, the result shows that dirty read is not possible in repeatable read isolation level. As discussed in section 2.2.1 a dirty read shows changes to other transactions, even if the changes are not committed yet.
-----DIRTY READ
########################################
Transaction t2:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t2
SELECT [EmployeeID]
  ,[Firstname]
  ,[Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
Current Lastname: Smith

########################################
Transaction t1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t1
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Carter'
WHERE [EmployeeID] = '1';

########################################
Transaction t1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t1
SELECT [EmployeeID]
  ,[Firstname]
  ,[Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
Current Lastname: Carter

########################################
Transaction t2:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t2
SELECT [EmployeeID]
  ,[Firstname]
  ,[Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
Current Lastname: Smith

Listing 11: Dirty read in repeatable read
3.3.3.2 Non-repeatable read

Non-repeatable read is not possible in repeatable read. As mentioned in section 2.2.2 this anomaly returns different results when performing the same selection query during a transaction. Listing 12 shows the steps to test against a non-repeatable read anomaly.

Listing 12: Non-repeatable read in repeatable read
3.3.3.3 Lost update
Lost update is not possible, because Snapshot isolation does not allow change attributes which are already changed by other transactions before. Listing 13 shows the steps and the corresponding error message when trying to provoke a lost update anomaly.
-- LOST UPDATE

Transaction t1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t1
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)

Current Lastname: Smith
Current Lastname: Springer

Transaction t1:
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION t1
UPDATE [ilv].[dbo].[persons] WITH (SNAPSHOT)
SET [Lastname] = 'Wayne'
WHERE [EmployeeID] = '1';

com.microsoft.sqlserver.jdbc.SQLServerException: The current transaction attempted to update a record that has been updated since this transaction started. The transaction was aborted.

Transaction t2:
COMMIT TRANSACTION t2;

Transaction t1:
COMMIT TRANSACTION t1;

Listing 13: Lost update in repeatable read
3.3.4 **Serializable**

3.3.4.1 **Dirty read**
Dirty read is not possible in serializable isolation level. Listing 14 shows the steps to provoke a dirty read. As the result shows, the UPDATE query doesn't change without a COMMIT.
OPTIMISTIC CONCURRENCY CONTROL an MULTIVERSION CONCURRENCY CONTROL with MS SQL Server 2014

Listing 14: Dirty read in serializable

```
-- Dirty read

Transaction t2:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t2
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Carter'
WHERE [EmployeeID] = '1';
EmployeeID: 1  Firstname: John  Lastname: Carter

Transaction t2:
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t1:
ROLLBACK TRANSACTION t1;

Transaction t2:
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
```

Listing 14: Dirty read in serializable
3.3.4.2 Non-repeatable read
Non-repeatable read is not possible in serializable isolation level. Listing 15 shows the steps to provoke a non-repeatable read. As the result shows the commit of t2 doesn't change the results of t1.

---
NON-REPEATABLE READ

Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t2:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t2
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Carter'
WHERE [EmployeeID] = '1';

Transaction t2:
COMMIT TRANSACTION t2;

Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Smith

Transaction t2:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t2
SELECT [EmployeeID], [Firstname], [Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)

EmployeeID: 1  Firstname: John  Lastname: Carter

Listing 15: Non-repeatable read in serializable
3.3.4.3 Lost update

Lost update is not possible in serializable isolation level. Listing 16 shows the steps to provoke a lost update. t2 cannot commit its changes and gets an error message instead.

```
-----------LOST UPDATE
########################################################
Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
SELECT [EmployeeID]
  ,[Firstname]
  ,[Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
EmployeeID: 1  Firstname: John  Lastname: Smith

########################################################
Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Wayne'
WHERE [EmployeeID] = '1';

########################################################
Transaction t2:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t2
UPDATE [ilv].[dbo].[persons] WITH(SNAPSHOT)
SET [Lastname] = 'Carter'
WHERE [EmployeeID] = '1';
com.microsoft.sqlserver.jdbc.SQLServerException: The current transaction attempted to update a record that has been updated since this transaction started. The transaction was aborted.

########################################################
Transaction t2:
COMMIT TRANSACTION t2;

########################################################
Transaction t1:
COMMIT TRANSACTION t1;

########################################################
Transaction t1:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRANSACTION t1
SELECT [EmployeeID]
  ,[Firstname]
  ,[Lastname]
FROM [ilv].[dbo].[persons] WITH (SNAPSHOT)
EmployeeID: 1  Firstname: John  Lastname: Wayne
```

Listing 16: Lost update in serializable
3.3.4.4 Write skew

Write skew is not possible in serialization isolation level, because commits will be executed in a serial order. Listing 17 shows how to provoke a write skew anomaly. Only the changes made by t1 were committed.

Listing 17: Write skew in serializable

3.4 Results

In this section we will take a look at the results of the test cases.
<table>
<thead>
<tr>
<th>Isolation Level/Anomalies (expected)</th>
<th>Dirty read</th>
<th>Non-repeatable read</th>
<th>Lost update</th>
<th>Write skew</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read committed / Snapshot isolation</td>
<td>impossible</td>
<td>possible</td>
<td>impossible (possible)</td>
<td>impossible (possible)</td>
</tr>
<tr>
<td>Repeatable read</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible (possible)</td>
<td>-</td>
</tr>
<tr>
<td>Serializable</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible</td>
<td>impossible</td>
</tr>
</tbody>
</table>

**Table 2: Results of the test cases and the anomalies**

The results of the test shows that in some cases the implemented isolation levels in memory-optimized tables differ from the definition in [1]. For example, Microsoft uses the snapshot isolation level as a hybrid of read committed and snapshot isolation defined in [1], this leads to a different behavior. The database detects when a transaction tries to UPDATE a data item, after another transaction changes that item. It then prevents the last transaction to modify the data so that lost updates are impossible, because a transaction is not able to commit.

Memory-optimized tables uses a technique of constraints which does not allow to validate input only on local snapshots, but on database, before a commit can be started. It is impossible to have a write skew anomaly, because of the serial checks during a transaction. Testing if an isolation level in memory-optimized tables allows write skew anomaly is impossible, because stored procedure in fact validate the current given value with another value on the database before commit. From architecture perspective it is not possible to get a write skew anomaly.

The main question is in which cases it is an advantage to allow or use anomalies. Dirty reads for example are not possible in memory-optimized tables. The reason is the architecture of this new tables. I.e. the logging mechanism only logs transactions which are committed. If every statement would be committed, which maybe has no effect on the tables, a large log file would be created and the memory would be used inefficient. Non-repeatable reads are possible in snapshot isolation. Preventing this anomaly can be beneficial for example when using a search with pagination. I still need the same results every time a query data, but going to page 4 of the search results. But allowing non-repeatable reads is a must for example in real-time environments such as watching stock markets or position of trains.
4 Conclusion and outlook

We have found out that OCC is an optimistic approach to manage concurrent transactions. The lock-free rule allows to execute transactions without blocking each other. OCC has one major disadvantage: write-write-conflicts needs to roll back the failed transaction and the commit process must be restarted. Read operations has no effects on other transactions. In fact the transaction rate is high, as long as no write-write-conflict occur.

MVCC is an implementation of OCC. It uses versioning to manage write operations and to prevent read operations fail for instance during an commit failure. When an UPDATE is executed on a data item, MVCC creates a new version of the data item instead of overwriting it. If there is no write-write-conflict the updated version of the data is available for the next read operation. The older version stays as long as a transaction needs it, otherwise it can be removed.

Microsoft SQL Server 2014 introduces Microsoft In-memory OLTP, which is realized with the “Hekaton” engine. The idea is to store data in main memory to improve performance. This goal was realized with the newly introduced memory-optimized table. This new type of table implements MVCC. Microsoft developed a new data structure for memory-optimized tables to allow the database to manage versions of rows. A garbage collector is responsible to remove old and unused versions of rows, to free memory if possible. The GC starts a background process to collect garbage versions that are not used by transactions and then removes pointers in index tables, after that the GC removes them from memory. The cooperative approach detects garbage versions during indexing progress.

Main memory in computers is limited, so caching mechanisms, pages or features which create would create more data are not recommended. Microsoft allows the using of hash indexes and nonclustered indexes to improve query performance.

The main problem with main memory is that it is not a persistent memory. If there is no power, all data on main memory get lost. Microsoft implements a hard-disk backup solution called Checkpoint. A Checkpoint is a file that contains all create transactions which are logged by a transaction log record. There is also logging for data creation and deletion, so that the checkpoint file only includes the data which were in the tables during the last stable state. When SQL Server 2014 recovers, the checkpoint file will be used to reconstruct the tables and data.

Memory-optimized table has limits in context of SQL isolation levels. It supports serializable, repeatable read, snapshot isolation and when some conditions are met read committed is supported to. This prevents database to allow dirty reads.

Disk-based tables and memory-optimized tables have more differences i.e. using data types, size of data structures, etc.

Unlike other In-memory database vendors, Microsoft adds In-memory OLTP as a feature and not as a standalone platform like SAP HANA or Oracle TimesTen In-memory database. Migration projects could be cheaper, because for example no extra licensing is needed. The worst case scenario is that a migration is not possible and the database schema must be redesigned, because the current design exceed the limitations of memory-optimized tables. Microsoft developed tools to support migrations which are also included in SQL Server 2014.
It is difficult to summarize how much performance gain with MVCC or memory-optimized tables. Microsoft arguments that it’s depending on the use case the speed can be raised from 5% to 20%. A benchmark by sqlauthority.com shows, that INSERT operations in memory-optimized tables performs about 50 times faster than disk-based tables. This improvements speed up data analyzing processes.

It is not possible to measure the performance of MVCC, but in theory the approach to never lock data items is quite faster, than blocking transactions like pessimistic concurrency control. Like said at the beginning of the paper, MVCC is especially recommended in cases conflicts are rare. Currently there is no efficient approach to manage write-write-conflict. It also exist in the implementation of SQL Server 2014. Microsoft recommends developers to manage these conflicts in their application. But the engine must handle the conflict and report it to the transaction owner or user or application. This costs time and resource. There is actually no solution. If such conflicts are mostly expected a pessimistic concurrency control approach should be chosen.

Technology researches from past, that were dig because there were no resources, are actually excavated, because now the resources are available. A big hype is actually “Big data”, which means to analyze large data and gain detailed information, support decision based on the information. For example it is now possible to store petabytes of data about a customer’s buying behavior and generate matching advertisements.

General Electric Co. (GE) for example tracks the mileage of airplane engines and turbines as part of maintenance. The collected data (approximately 500GB per airplane per day) is used to forecast failures (Source [14]). With traditional databases analyzing such large (big) data would take a lot of time. Modern In-memory databases bring more performance and new possibilities to develop innovations from the past, which were not able to develop or use.

MVCC does not lock data items, so that read transactions do not influence each other. The disadvantage is when write-write-conflicts occur, because after an error a rollback is done and the transaction has to be recreated. Another disadvantage is the use of a garbage collector, which analyzes which versions of rows are no longer needed and delete them. Because of these problems it is not sure if MVCC will be used in future In-memory databases. But common database systems like PostgreSQL, solidDB by IBM, TenTimes by Oracle and of course SQL Server 2014 by Microsoft are already using MVCC.

New hardware will be developed and new possibilities and as well new approaches will be researched.
5 Bibliography


6 Figures

Figure 1: Dirty read example .................................................................................................................. 8
Figure 2: Non-repeatable read example .................................................................................................... 8
Figure 3: Lost update example ................................................................................................................ 9
Figure 4: Write skew example .................................................................................................................. 9
Figure 5: Row in multiversion concurrency control ................................................................................. 13
Figure 6: Write-write-conflict ................................................................................................................... 13
Figure 7: The structure of a row in a memory-optimized table, Source: [8], “Rows” ....................... 15
Figure 8: A hash index with a single row, Source: [8] ............................................................................. 18
Figure 9: A hash index with two rows, Source: [8] .................................................................................. 19
7 Tables

Table 1: Isolation Levels and their anomalies ................................................................. 10
Table 2: Results of the test cases and the anomalies ....................................................... 40
8 Listings

Listing 1: CREATE TABLE operation with a hash index, Source: [8] .......................... 18
Listing 2: CREATE TABLE operation with a nonclustered index, Source: [7] .................. 19
Listing 3: SELECT operation with range, Source: [10] ........................................ 20
Listing 4: person table ......................................................................................... 23
Listing 5: Table accounts .................................................................................... 24
Listing 6: Stored procedure to validate amount1 and amount2 ............................ 25
Listing 7: Dirty read in snapshot isolation .......................................................... 27
Listing 8: Non-repeatable read in snapshot isolation .......................................... 28
Listing 9: Lost update in snapshot isolation ......................................................... 29
Listing 10: Write skew in snapshot isolation ........................................................ 30
Listing 11: Dirty read in repeatable read .............................................................. 31
Listing 12: Non-repeatable read in repeatable read ............................................ 32
Listing 13: Lost update in repeatable read .......................................................... 34
Listing 14: Dirty read in serializable ................................................................... 36
Listing 15: Non-repeatable read in serializable .................................................... 37
Listing 16: Lost update in serializable ................................................................. 38
Listing 17: Write skew in serializable .................................................................. 39