

Konzeption und prototypische Implementierung eines leistungsoptimierten Archivierungssystems

Conception and prototypical implementation of a
performance-optimized archiving system

Leonard Tutzauer

Master-Abschlussarbeit

Betreuer: Prof. Dr. Christoph Schmitz

Trier, 18.08.2019

Vorwort

Die vorliegende Masterarbeit „Konzeption und prototypische Implementierung eines leistungsoptimierten Archivierungssystems entstand in Zusammenarbeit mit dem Fachbereich Wirtschaftsinformatik der Hochschule Trier und in Kooperation mit dem Unternehmen Köhl MBAG in Luxemburg.

An dieser Stelle möchte ich jedem aus der Abteilung Logistik- und Informationssysteme der Firma Köhl MBAG, der durch seine fachliche Kompetenz und persönliche Unterstützung zu dieser Arbeit beigetragen hat, danken. Ein ganz besonderer Dank gilt den folgenden Personen aus dieser Abteilung:

Technical Director Arno Fries, Michael Kohl, Lukas Vogel, Raphael Matter und Heiko Polgert, denn ohne Euch wäre dieses Projekt nicht machbar gewesen.

Weiter gilt ein ganz besonderer Dank meiner Freundin Laura Suss und meinen Eltern, Genet und Richard Tutzauer die mich in meinen Entscheidungen unterstützt haben und in den richtigen Situationen der Wind in meinen Segeln waren, den ich gebraucht habe.

Darüber hinaus will ich auch einem damaligen Kommilitonen und jetzt gutem Freund, Christoph Braun, danken. Nicht nur stand er zu jeder Zeit hinter mir, sondern war eine helfende Hand als meine Laptopplatine durchgebrannt ist und die Daten wiederhergestellt werden mussten.

Zu guter Letzt möchte ich meinem Betreuer Prof. Dr. C. Schmitz danken, der sich dem Thema gestellt hat und immer ein offenes Ohr für neue Ansätze und Lösungsvorschläge hatte. Sogar nachts und an Sonntagen sträubte er sich nicht davor, mir meine Fragen per E-Mail zu beantworten.

Kurzfassung

Angesichts des stetig wachsenden Datenvolumens, das in einem Unternehmen zu bewältigen ist, ist nicht mehr die Speicherkapazität ein Problem, sondern eher der performante Zugriff auf die Daten.

Da die Daten nicht immer gleich oft abgerufen werden, wird in der Regel ein Archiv als Hilfsmittel benutzt, das Daten, die vergleichsweise wenig abgerufen werden, enthält. In einem System, das mit Objekten arbeitet und somit auch viele Abhängigkeiten aufweist, ist die Aufteilung in Produktiv- und Archivdaten nicht selbsterklärend.

In dieser Arbeit geht es darum, Methoden auszuarbeiten, die ein bestehendes System in seiner Performanz optimieren. Dazu werden zunächst Methoden zur Steigerung der Performanz durch Anpassung des Designs ausgearbeitet und verschiedene Archivierungsmethoden untersucht. Zudem wird die Partitionierung als Archivierungsmethode prototypisch entwickelt.

Die Partitionierung bewirkt bei Einhaltung bestimmter Kriterien eine bis zu zehnfache Performanzsteigerung, während bei deren Nichteinhaltung die Performanz stagniert.

Die entwickelten Methoden zeigen, dass die Partitionierung in der Praxis geeignet ist, um die Schnelligkeit beim Abrufen von Daten um ein Vielfaches zu steigern. Zusätzlich erweist sich die Partitionierung als ressourcensparend, da deren Implementierung nur auf Datenbankebene stattfindet und keine Software-Anpassung erforderlich ist.

Abstract

With the ever-growing amount of data to be stored in a company, storage capacity is no longer a problem, but rather the high-performance access to the data.

Since the data are not always retrieved the same number of times, an archive is usually used as an aid. It contains data that are comparatively not often accessed. In a system that works with objects and thus has many dependencies, the division into productive and archive data is not self-explanatory.

This thesis is about elaborating methods that optimize an existing system in its performance. First, methods for improving performance, by adapting the design are discussed and various archiving methods are examined. Ultimately, partitioning is prototypically developed as a method of archiving.

Partitioning increases performance by up to the tenfold if certain criteria are met, while performance is consistent if not adhered to.

The result shows that partitioning is a practical method that can increase the speed of retrieving the data many times over. In addition, since the implementation of partitioning only happens at the database level and no software customization is needed, this method proves to be resource efficient.

Inhaltsverzeichnis

1	Einleitung und Problemstellung	1
2	Grundlagen	3
2.1	Datenbanken	3
2.1.1	ACID-Prinzip	4
2.1.2	Datenbanksprache	5
2.1.3	Relationale Datenbank.....	6
2.1.4	Objektdatenbanken	7
2.1.5	Dokumentenbasierte Datenbank (NoSQL)	7
2.2	Objektrelationale Abbildung.....	8
2.2.1	Java-Persistenz-API (JPA).....	9
2.2.2	Hibernate.....	10
2.2.3	Anwendungsbeispiel Hibernate	11
2.3	Partitionierung.....	14
2.3.1	Partitionsfunktion	15
2.3.2	Partitionsschema.....	15
2.3.3	Indexierung und Datenausrichtung	16
2.3.4	Partitionsfunktionen	17
2.3.5	Anwendungsbeispiel.....	18
2.3.6	Sliding-Window	24
3	Ziele und Anforderungen	26
3.1	Ziele.....	26
3.2	Anforderungen	26
4	Performanzsteigerung	29
4.1	Designanpassung.....	29
4.1.1	Index	29
4.1.2	Materialisierte- und indizierte Sichten	30
4.1.3	Datenbankdesign	32
4.1.4	Wahl der Datenbank	35
4.2	Softwaretuning	39
4.2.1	Hibernate Optimierung	40
4.2.2	SQL Server-Konfiguration.....	47
5	Archivierungsverfahren	50
5.1.1	Logische Archivierung	51
5.1.2	Eine Datenbank mit zusätzlichen Archivtabellen	51
5.1.3	Nutzung von Datenbanken für Produktiv- und Archivdaten	51
6	Proof-of-Concept	56
6.1	Entwicklungsumgebung	56
6.2	Erstellung des Datenmodells	56

6.3	Umsetzung der Partitionierung	61
7	Test und Vergleich.....	64
7.1	Performanztest	65
7.2	SQL-Ausführungsplan.....	68
8	Zusammenfassung und Ausblick.....	70
	Literatur	71
	Anhang.....	76
	Erklärung der Kandidatin / des Kandidaten.....	82

Abbildungsverzeichnis

Abbildung 1: Struktur von Objekten in einem Dokument.....	8
Abbildung 2: Abbildung von Entitäten in eine Tabelle.....	9
Abbildung 3: Abfragebeispiel in Hibernate Query Langage	10
Abbildung 4: Auszug aus der Hibernate-Datei persistence.xml	12
Abbildung 5: Erzeugung eines EntityManagers mittels der EntityManagerFactory	13
Abbildung 6: Klasse Customer.java mit Annotationen	13
Abbildung 7: Codeausschnitt einer Klasse mit einer One-to-Many Beziehung	13
Abbildung 8: Partitionsfunktion.....	15
Abbildung 9: Partitionsschema	16
Abbildung 10: Erstellung einer Tabelle unter Anwendung des Partitionsschemas	16
Abbildung 11: Partitionsfunktion – Split.....	17
Abbildung 12: Filegroup via Quellcode erstellen	17
Abbildung 13: Alter Partition Scheme Next Used	17
Abbildung 14: Partitionsfunktion – Merge.....	18
Abbildung 15: Partitionsfunktion – Switch	18
Abbildung 16: Filegroups (FG1, FG2, FG3) erstellen	19
Abbildung 17: Partitionsfunktion und Partitionsschema angewendet.....	20
Abbildung 18: Tabelle mit Datensätzen befüllen.....	20
Abbildung 19: Ansicht mit Informationen einer Partition (Doward, 2019)	21
Abbildung 20: Erstellung der View PartitionInfo (Randal, Tripp, & Delaney, 2009).....	22
Abbildung 21: Ausgabe der View PartitionInfo	22
Abbildung 22: Partition Next Used und Split-Partition.....	22
Abbildung 23: Rückgabe der View nach Erstellung der FG4 und dem Einfügen von Daten ..	23
Abbildung 24: Merge-Funktion	23
Abbildung 25: Rückgabe nach Zusammenführung von Daten mittels Merge-Operators	23
Abbildung 26: Partitionsfunktion Switch: Partition 1 in eine leere Tabelle.....	24
Abbildung 27: Rückgabe der View nach Nutzung des Switch-Operator	24
Abbildung 28: Schieben der Daten von myOrderTableTMP zu myOrderTableDestination ...	24
Abbildung 29: Truncate-Funktion mit Angabe der Partition.....	25
Abbildung 30: Materialized-View erstellen in Oracle (Oracle Help Center, n.d.)	31

Abbildung 31: Datenbank-Entwurfs-Zyklus.....	32
Abbildung 32: Leistung Hibernate (grün) im Vergleich zu anderen JPA/DBMS-Kombinationen (grau) (http://www.jpab.org/Hibernate.html).....	37
Abbildung 33: Leistung der Objekt-Datenbank ObjectDB (grün) im Vergleich zu anderen JPA/DBMS-Kombinationen (grau) (http://www.jpab.org/ObjectDB.html)	38
Abbildung 34: Dokument einer dokumentbasierten Datenbank (http://wiki.de/lib/exe/fetch.php?cache=&media=bigdata:documentstore.png)	39
Abbildung 35: Optimistic-Locking mittels Version.....	40
Abbildung 36: Optimistic-Locking mittels Timestamp.....	40
Abbildung 37: AllocationSize (Wilming, 2013).....	41
Abbildung 38: Syntax Eager- und Lazyloading (Wilming, 2013).....	42
Abbildung 39: First-Level-Cache, Session-Objekt (https://cdn2.howtodoinjava.com/wp-content/uploads/hibernate_first_level_cache.jpg).....	42
Abbildung 40: Second-Level-Cache (eigene Darstellung, in Anlehnung an Abbildung 42) ...	43
Abbildung 41: Fetch vor Hibernate 5.1 (Jansen, Thoughts On Java, 2019).....	44
Abbildung 42: Fetch nach Hibernate 5.1 (Jansen, Thoughts On Java, 2019).....	44
Abbildung 43: Daten aus First-Level-Cache lesen, vor Datenbankzugriff (Jansen, Thoughts On Java, 2019)	45
Abbildung 44: Zwei Entitäten ohne Relation mittels kartesischen Produkts verbinden (Jansen, Thoughts On Java, 2019).....	45
Abbildung 45: Zwei Entitäten ohne Relation mittels Join-Operator verbinden (Jansen, Thoughts On Java, 2019).....	45
Abbildung 46: Select-Anfragen N+1.....	46
Abbildung 47: HQL Fetch-Join und wie diese in SQL interpretiert wird.....	46
Abbildung 48: Lösung des N+1-Problems mittels Kriterien a) und Fetch-Join b) (Suda, 2014)	46
Abbildung 49: Letzten guten Plan forcieren (https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/media/force-last-good-plan.png?view=sql-server-2017)	48
Abbildung 50: Union-Operator über zwei Tabellen in zwei unterschiedliche Datenbanken ...	52
Abbildung 51: Update, Delete-Prozedur	54
Abbildung 52: Klassendiagramm.....	57
Abbildung 53: Ausschnitt aus der Klasse OrderItemCompositeKey.java	58
Abbildung 54: Orders.java.....	59
Abbildung 55: Inhalt der Spalte UnitPrice der Tabelle OrderItem aktualisieren	60

Abbildung 56: Inhalt der Spalte TotalAmount der Tabelle Orders aktualisieren	60
Abbildung 57: Datenmodell.....	61
Abbildung 58: Partitionsfunktion und Partitionsschema	61
Abbildung 59: Die Schritte Löschen sowie Setzen der Schlüssel zur Partitionierung.....	62
Abbildung 60: Validierung der Partitionierung	63
Abbildung 61: Ergebnis Query Nummer 3.....	67
Abbildung 62: Ergebnis Query Nummer 2.....	68
Abbildung 63: Ausführungsplan der Query Nummer 3	69
Abbildung 64: Ausführungsplan der Query Nummer 4	69
Abbildung 65: Anhang: Sliding Window gekapselt in einer Stored Procedure	76
Abbildung 66: Anhang: Erstellung einer Indexed-View und Nutzung ohne explizite Nennung (Microsoft SQL-Dokumentation, 2018)	77
Abbildung 67: Anhang: Durchschnittsergebnisse der Tabelle: 19 dargestellt in Diagramme ..	81

Tabellenverzeichnis

Tabelle 1: Data Definition Language (DDL).....	5
Tabelle 2: Data Manipulation Language (DML)	6
Tabelle 3: Data Control Language (DCL)	6
Tabelle 4: Beispiele für Annotationen (Yatin, 2017)	11
Tabelle 5: CRUD-Operationen und deren Umsetzung in SQL und Java	11
Tabelle 6: Parameter Hibernate.hbm2ddl.auto (Hibernate Community Documentation, n.d.)	12
Tabelle 7: Methoden des EntityManagers	14
Tabelle 8: Verhalten der Datensätze bei Range LEFT und Range RIGHT	15
Tabelle 9: Ziele.....	26
Tabelle 10: Funktionale Anforderung Schablone	27
Tabelle 11: Anforderungen	28
Tabelle 12: Die fünf Normalformen.....	34
Tabelle 13: Parallele Ausführung vs. serielle Ausführung (Yaseen, MSSQLTips, 2017)	49
Tabelle 14: Technische Daten der Testumgebung	64
Tabelle 15: Anzahl der Datensätze in den Datenbanken und deren Größe in Gigabyte	65
Tabelle 16: Nummer und Aufgabe der entsprechenden Anfrage.....	66
Tabelle 17: Nummer und die Anfrage in der SQL-Sprache	66
Tabelle 18: Zusammenfassung des Performanztests (Durchschnittswerte)	67
Tabelle 19: Anhang: Queries aus Kapitel Test und Vergleich.....	78
Tabelle 20: Anhang: Ergebnisse in ms aus Test und Vergleich	80

1 Einleitung und Problemstellung

In der heutigen Zeit wird nahezu alles digitalisiert. Demzufolge werden immer größere Datenspeicher benötigt, um die Daten persistent zu bewahren (Radtke, 2019). Bei der wachsenden Anzahl an Daten ist jedoch nicht nur die Speicherung der Daten zu einem Problem bzw. Kostenfaktor geworden, sondern vorrangig Möglichkeiten, auf sie zuzugreifen (Bauer, 2018), denn Daten, die auf großen Speichermedien lagern, müssen grundsätzlich auch zu jeder Zeit und vor allem unter einer geringen Latenz abrufbar sein.

Um den Kostenfaktor zu senken und das Problem des ständig wachsenden Datenvolumens in den Griff zu bekommen, ist es üblich, die Daten auf mehrere Datenmedien zu verteilen (Veikko Krypczyk, 2018). In der Regel werden Daten unterschiedlich oft abgerufen. Aus diesem Grund werden Daten, die nicht oft verwendet werden, in einem Archiv gespeichert. Dabei ist es vom Kostenfaktor her günstiger, mehrere Medien zu nutzen und diese zu einem Verbund als logische Einheit zusammenzuführen, als ein großes Speichermedium vorzuhalten. Zudem ist bei einem partiellen Ausfall eines Speichermediums der Zugriff auf die Speichermedien, die nicht vom Ausfall betroffen sind, in der Regel immer noch gegeben. Ein weiterer Vorteil besteht darin, dass Zugriffe auf diese Daten nicht die Performanz von nur einem Rechner zur Verfügung haben, sondern die Performanz aus einer Kombination aus allen Rechnern des Verbunds besteht. Somit können Anfragen schneller durchgeführt werden. Zudem ist die Wahrscheinlichkeit geringer, dass eine Maschine von der Performanz her an ihre Grenzen stößt, da die Last der Zugriffe auf verschiedene Systeme verteilt wird (Lipinski, 2016).

Die Abteilung Logistik- & Informationssysteme (LIS) der Firma Köhl MBAG hat sich auf die Realisierung standardisierter Manufacturing-IT-Lösungen in dem Bereich Intra- und Produktionslogistik spezialisiert. Das Unternehmen verwendet eine Software, um den Materialfluss und die Lagerverwaltung zu kontrollieren. Dabei sollen historische Daten zur Laufzeit archiviert und Daten über Webservices für Kunden bereitgestellt werden.

Im Rahmen der Masterarbeit und in Kooperation mit der Firma Köhl werden verschiedene Methoden erarbeitet, die ein bereits bestehendes System in seiner Performanz optimieren sollen. Die Performanz kann zum einen durch die Anpassung des Designs erhöht werden und zum anderen durch eine passende Archivierungsstrategie, die anschließend prototypisch entwickelt und implementiert wird. Die vorgegebenen Anforderungen an das Archivierungstool beinhalten, dass das Archivieren und Auswerten der Daten die Performanz des Produktivsystems nicht belastet. Ebenfalls soll eine Option vorhanden sein, mit der das Abrufen von Daten und Einbinden von Archivdaten möglich ist.

Die Basissoftware der Firma Köhl arbeitet auf einer Java-EE¹-Architektur und benutzt den JBoss-Application-Server EAP² von Red Hat, um Zugriffe über das Internet anzubieten. Der Application-Server bringt das Open-Source-Framework Hibernate von Haus aus mit, das genutzt wird, um Plain Old Java Objects (POJOs) inklusive Relationen (1...n und n...m) in die relationale Datenbank von Microsoft persistent zu speichern. Zudem hat es die Aufgabe, mit den in der relationalen Datenbank hinterlegten Datensätzen die ursprünglichen Objekte zu

¹ Java EE: Java Enterprise Edition

² EAP: Enterprise Application Platform

rekonstruieren. Dieser Vorgang wird als objektrelationales Mapping bezeichnet. Hibernate ist die Implementierung der JPA³-Spezifikation, die einen allgemeinen Standard zur Lösung der objektrelationalen Abbildung (Lipinski, ITWissen.info, 2013) darstellt.

Das Unternehmen nutzt die Datenbank von Microsoft (SQL Server), auf der alle Daten gespeichert werden. Diese Daten lassen sich in Produktivdaten und historische Daten klassifizieren. Produktivdaten sind dabei die Daten, die täglich gebraucht werden. Historische Daten können z. B. Tracking-Daten sein. Diese werden abgespeichert und in der Regel nur in spezifischen Ausnahmefällen abgerufen. Durch die Art der Speicherung der Daten kann es passieren, dass Abfragen unter anderem über Daten laufen, die irrelevant sind.

Da nun historische Daten und Produktivdaten auf einer Datenbank liegen, ist es möglich, dass eine Anfrage historische Daten abrufen, obwohl diese für die Rückgabe irrelevant sind. Dies führt dazu, dass die Rückgabe der Anfragen länger dauert und mehr Performanz-Ressourcen in Anspruch nimmt. Zudem gibt es diverse Benutzergruppen, u. a. die Produktion, die mit der Datenbank arbeiten. Die Produktion benötigt die Daten möglichst schnell, um eine zuverlässige Herstellung und termingerechte Fertigung zu gewährleisten. Erfolgen nun weitere Zugriffe auf die Datenbank durch andere Benutzergruppen, kann das dazu führen, dass das System langsamer arbeitet und der Herstellungsprozess stagniert.

Bei der Integrierung eines Archives ist auf die 1...n- und m...n-Beziehungen zwischen den einzelnen Entitäten der Daten zu achten. Da man die Relationen und die Entitäten nicht getrennt voneinander betrachten kann, müssen Entitäten zusammen mit ihren Relationen im Archiv gespeichert werden. Anderenfalls können die Entitäten bei ihrer Wiederherstellung nicht mehr entsprechend zuordnen können. Zudem geht diese Art der Archivierung infolge der Rahmenbedingung, die besagt, dass man Datensätze im Archiv verändern kann, über das gewöhnliche Archivierungsproblem hinaus, denn in der Regel dient ein Archiv lediglich zum Speichern von Daten und Editieren ist dort nicht gestattet. Eine weitere Rahmenbedingung fordert, dass Entitäten anhand eines Datums im Archiv automatisiert gespeichert bzw. aus dem Archiv gelöscht werden. Ein weiteres Problem bei der Integration eines Archivs ist die Delegation der Anfragen zwischen Archiv und Produktivsystem. Benutzer sollen die Möglichkeit haben, Daten abzurufen, ohne die darunterliegende Struktur zu kennen. Somit muss eine Instanz existieren, die die Anfrage der Benutzer verarbeitet und das richtige Ergebnis liefert.

³ JPA: Java Persistence API

2 Grundlagen

In diesem Kapitel werden notwendige Grundlagen beschrieben, die zu einem besseren Verständnis der Arbeit verhelfen. Untergliedert ist dieses Kapitel in drei Punkte. Im ersten Unterpunkt – Datenbanken – wird erläutert, was Datenbanken sind und wie diese funktionieren. Zudem werden für diese Arbeit relevante Datenbankmodelle vorgestellt. Im folgenden Unterkapitel wird die Problematik von objektrelationalen Abbildungen im Zusammenhang mit der Spezifikation Java Persistence Api, die von Hibernate implementiert wird, beschrieben. Im letzten Unterkapitel wird die Partitionierung vorgestellt, die am Ende der Arbeit realisiert wird.

2.1 Datenbanken

Um große Mengen an Daten widerspruchsfrei, effizient und dauerhaft zu speichern, verwendet man Datenbanken. Diese Daten können dann in unterschiedlicher Form aufgeteilt und kombiniert werden, sodass sie für den Benutzer bzw. für die Software im effizientesten Zustand bereitstehen. Eine Datenbank oder genauer gesagt ein Datenbanksystem (DBS) bezeichnet ein Datenverwaltungssystem und besteht aus zwei Bereichen. Das Datenbankmanagementsystem (DBMS) ist die eigentliche Verwaltungssoftware und bildet den ersten Teil des Datenbanksystems. Dieses System verwaltet, wie die Daten intern gespeichert werden. Zudem werden lesende und schreibende Zugriffe überwacht und kontrolliert. Den zweiten Teil bilden die eigentlichen Daten in der Datenbank. Man spricht hier auch von einer Datenbasis.

Die Kommunikation zwischen Benutzer, Programm und einem Datenbanksystem erfolgt durch die Nutzung einer Datenbanksprache. Diese Datenbanksprache wird von einem Datenbanksystem angeboten. Im Folgenden werden Anforderungen an eine Datenbank aufgelistet:

Anforderungen an ein DBMS:

- Daten speichern, editieren und löschen
- Einsatz von Triggern⁴ und gespeicherten Prozeduren⁵
- Metadaten verwalten
- Gewährleistungen der Sicherheitsvorkehrungen in puncto Datenschutz, Datenintegrität und Datensicherheit
- Transaktionsprinzip: Änderungen sollen entweder vollkommen oder gar nicht vollzogen werden
- Möglichkeit, Abfragen zu verbessern

⁴ Ein Mechanismus, der durch bestimmte Befehle ausgelöst wird (Microsoft SQL-Dokumentation, 2019)

⁵ Eine oder mehrere gespeicherte Anweisungen, die Eingabeparameter besitzen und Werte zurückliefern können (Microsoft SQL Dokumentation, 2017)

Es gibt unterschiedliche Arten von Datenbanken. Der Unterschied liegt unter anderem in der Art und Weise, wie Daten in einem System gespeichert werden. Das Datenmodell, das durch das DBMS festgelegt wird, ist die theoretische Grundlage dafür, wie Daten gespeichert werden. Es ist das Grundgerüst eines Datenbanksystems (Gebhardt, 2018).

Das relationale Datenbanksystem ist das bekannteste der Modelle. Jedoch gibt es neben der relationalen unter anderem das objektorientierte Datenbanksystem (OODBMS) und die dokumentenbasierte Datenbank (DB-Engines, 2019).

2.1.1 ACID-Prinzip

Die meisten Datenbanken folgen den Grundregeln des ACID-Prinzips. Dies sind Regeln und Eigenschaften, die die Verlässlichkeit und Konsistenz der Transaktionen und der damit verbundenen Datenbankmanagementsysteme gewährleisten. ACID ist ein Akronym und steht für die englischen Begriffe [a]tomicity, [c]onsistency, [i]solation und [d]urability, gebräuchlich ist auch das deutsche AKID⁶ (Luber, 2018). Die Grundprinzipien sind wie folgt beschrieben:

Atomarität: Eine Sequenz einzelner, aufeinander folgender Aktionen bezeichnet man als eine Transaktion. Dabei müssen entweder alle Einzelaktionen vollkommen oder überhaupt nicht ausgeführt werden. Im Falle einer Fehlermeldung im Laufe der Sequenz muss das System dafür sorgen, dass alle bereits getroffenen Änderungen rückgängig gemacht werden. Dies bedeutet, dass alle Spuren einer fehlgeschlagenen Transaktion komplett aus der Datenbank gelöscht werden müssen. Alle vorgenommenen Änderungen sind also erst gültig, wenn die komplette Sequenz durchgelaufen ist. In Datenbanken wird die Atomarität gesichert, indem ein ausführliches Logging protokolliert wird.

Konsistenz: Eine Datenbank sollte sich nach jeder vollzogenen Aktion wieder in einem konsistenten Zustand befinden. Dies bedeutet, dass die datenbankspezifischen definierten Bedingungen der Integrität und die logische Konsistenz der Daten vor dem Abschluss einer jeden Transaktion geprüft werden müssen. Beispielsweise wäre die Einhaltung bestimmter Wertebereiche eine solche datenbankspezifische Bedingung. Sollte eine Transaktion gegen die festgelegten Konsistenzbedingungen verstoßen, so wird diese abgelehnt, die Änderungen der Transaktion werden ungültig und die Daten wieder in ihren ursprünglichen Zustand zurückgesetzt. Im Laufe der Transaktion darf sich die Datenbank im inkonsistenten Zustand befinden, doch vor und nach jeder Transaktion muss die Konsistenz gesichert werden.

Isolation: Datenbanken sind eine Partei im Prozess der Datensicherung, doch sind auch andere Parteien, wie zum Beispiel Benutzer, oder Prozesse notwendig. Aus diesem Grund ist es üblich, dass mehrere Benutzer gleichzeitig an den Daten arbeiten beziehungsweise mehrere Prozesse innerhalb der Datenbank zeitgleich verlaufen. Dies könnte theoretisch zu negativen Auswirkungen wie gegenseitigem Überschreiben oder Löschen einzelner Daten führen. Isolation (Abgrenzung) wirkt den negativen Auswirkungen entgegen, sie stellt sicher, dass jeder Benutzer die Datenbank wahrnimmt, als wäre er der einzige Benutzer. Somit bleiben die Änderungen der anderen Benutzer für einen selbst unsichtbar und auch die Transaktionen beeinflussen sich nicht gegenseitig. Isolation wird innerhalb eines Datenbanksystems von Sperrverfahren gesichert. Sind alle Voraussetzungen erfüllt, ist die Integrität des Ablaufs gewährleistet.

⁶ Ist ein Akronym und steht für Atomarität, Konsistenz, Isolation und Dauerhaftigkeit

Dauerhaftigkeit: Der letzte Aspekt des ACID-Prinzips ist die Dauerhaftigkeit. Die Daten müssen sicher und dauerhaft in der Datenbank gespeichert sein, sodass kein Systemfehler oder -absturz ein unwiderrufliches Löschen der Daten bewirkt. Verlorene Daten müssen wiederherstellbar sein, was mithilfe von Logging Maßnahmen erreicht werden kann. Ein Transaktionslog stellt alle Informationen sicher und garantiert deren Verfügbarkeit für den Benutzer, damit Daten sogar nach einem Systemausfall rekonstruiert werden können.

Die Einhaltung der ACID-Prinzipien bringt für Benutzer wie auch für Entwickler viele Vorteile mit sich. Alle Beteiligten profitieren von einer fehlerfreien Arbeitsumgebung und konsistenten Datenaufbewahrung. Wenn die Arbeitsschritte korrekt ausgeführt werden, wie im Falle einer vollständigen Transaktion, so ist auch die Dauerhaftigkeit der Daten gegeben. Gleichzeitig ist das System auch gegen Fehler abgesichert, so führen Fehlermeldungen innerhalb einer Transaktion nicht zu Datenveränderungen oder gar Verlusten. Dieser Punkt hat zur Folge, dass keine zeitaufwendige Recherche zur Fehlerbehebung nötig ist. In Systemen, die mehrere Benutzer zulassen, garantiert das ACID-Prinzip reibungslose, parallele Arbeit (Luber, 2018).

Jedoch bringt das ACID-Prinzip auch Nachteile mit sich, denn um die Daten konsistent zu halten, können sie von jedem, der die Berechtigung hat, gelesen werden. Gleichzeitig aber wird die Verarbeitung der Daten eingeschränkt. Fachsprachlich heißt das, dass Daten gelockt werden. Die Art und Weise, wie die Daten gesperrt werden, ist Einstellungssache. Die feinste Granularität wäre es, Daten nur von einem einzigen Befugten ändern zu lassen. Das würde aber dazu führen, dass Benutzer auf diese Daten während einer Änderung nicht zugreifen könnten und somit warten müssten, bis das geforderte Ziel wieder verfügbar ist.

Wenn nun Benutzer B1 eine Änderung am Objekt X vollzieht, das Daten von Objekt Y braucht und ein anderer Benutzer B2 im gleichen Moment Objekt B verändert, das wiederum Daten von Objekt A bezieht, entsteht ein „Deadlock“, denn durch B1 wird X gelockt und durch B2 wird Y gelockt. Da X und Y jeweils Daten des anderen benötigen, diese aber durch die B1 und B2 gesperrt sind, entsteht hier eine Verklemmung (Barry, 2019). Um sie zu beheben, müssten weitere Mechanismen eingebaut werden, beispielsweise das Zwei-Phasen-Protokoll (Goram, 2017).

2.1.2 Datenbanksprache

Die Datenbanksprache wurde entwickelt, damit Benutzer oder Programme mit dem Datenbanksystem kommunizieren können. Mit dieser Sprache werden ausschließlich Abfragen formuliert, um Datenstrukturen oder Daten aus der Datenbank zu bearbeiten.

Eine standardisierte Datenbanksprache für relationale Datenbanken ist SQL. Gestellte Anfragen werden nicht von SQL kontrolliert, das heißt, es wird nicht untersucht, wie der Code zu implementieren ist, sondern nur, wie die Datenbank sich bei bestimmten Eingaben verhalten soll. Unterstützt wird SQL in unterschiedlichen Formen und Variationen und zum Teil leicht angepasst von fast allen gängigen Datenbanksystemen. Bei einer Variation von SQL spricht man von einem Dialekt. Datenbanksprachen werden in Datenbeschreibungssprache, Datenverarbeitungssprache und Datenaufsichtssprache klassifiziert (Gebhardt, 2018).

Die Datenbeschreibungssprache oder auch Data Definition Language (DDL) ist die Sprache, die verwendet wird, um Datenstrukturen anzulegen, zu verändern und zu löschen. In Tabelle 1 ist ein Auszug der DDL in Microsoft SQL Server zu sehen (Vuk & Geelen, 2016).

Aktion	Ausführung
Erstellen	[CREATE] [Database Table View] Name
Löschen	[DROP] [Database Table View] Name
Ändern	[ALTER] [Database Table View] Name

Tabelle 1: Data Definition Language (DDL)

Die Datenverarbeitungssprache, auch Data Manipulation Language (DML) genannt, ist die Sprache, die verwendet wird, um Datenbestände oder Sprachteile einzufügen, zu ändern, zu löschen oder abzufragen (Vuk & Geelen, 2016). In Tabelle 2 wird ein Beispiel der DML in Microsoft SQL Server dargestellt.

Aktion	Ausführung
Abfragen	<code>SELECT column1, column2, ... FROM table_name;</code>
Einfügen	<code>INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);</code>
Ändern	<code>UPDATE table_name SET column1 = value1, column2 = value2, ... WHERE condition;</code>
Löschen	<code>DELETE FROM table_name WHERE condition;</code>

Tabelle 2: Data Manipulation Language (DML)

Die Datenaufsichtssprache, engl. Data Control Language (DCL), ist der dritte und letzte Bestandteil der SQL-Sprache. Dieser Teil der Sprache ist für die Zugriffskontrolle und Rechtevergabe zuständig. Manchmal wird DCL nicht als eigenständige Komponente der Sprache SQL gesehen, sondern als eine Teilmenge von DDL (Vuk & Geelen, 2016). In Tabelle 3 befindet sich ein Auszug der DCL in Microsoft SQL Server.

Aktion	Ausführung
Rechte vergeben	<code>Grant [select] [insert] [update][delete] on tabelle1 to person1</code>
Rechte entziehen	<code>Revoke insert on tabelle1 to person1</code>
Rechte verbieten	<code>deny update on tabelle1 to person1</code>

Tabelle 3: Data Control Language (DCL)

2.1.3 Relationale Datenbank

Das relationale Datenbankmodell ist am weitesten verbreitet und wird in der Datenbankentwicklung als Standard genutzt. Das Fundament des Datenbankmodells besteht aus vier Elementen: Tabellen, Attributen, Beziehungen und den Grundlagen der relationalen Algebra. Sie stellt eine mathematische Beschreibung einer Tabelle und ihre Beziehung zu anderen möglichen Tabellen dar. Die Operationen auf diesen Relationen werden durch die relationale Algebra bestimmt.

Das Relational-Database-Management-System (RDBMS) wird dem Datenbankmanagementsystem zugeordnet. Es wurde von Edgar F. Codd 1970 erstmals beim Softwareunternehmen IBM vorgestellt (Luber S., 2017). Das RDBMS nutzt die Sprache SQL, um Daten oder Strukturen der Tabelle und der Architektur abzufragen und zu manipulieren.

Ein Merkmal dieses Datenbankmodells ist es, Daten in mehrere Tabellen zu speichern, die aus Spalten und Zeilen bestehen. Die Tabellen werden miteinander durch Primär- und Fremdschlüssel verbunden. Eine Zeile ist ein Datensatz, enthält Daten von mehreren Spalten und kann somit auch als Tupel angesehen werden. Dabei sind die Spalten als Attribute der Tabelle zu betrachten. Attribute können von einem bestimmten (Daten-)Typ wie Integer, Strings, Date, Varchar usw. sein.

Das Grundprinzip der relationalen Datenbank besteht darin, dass die Daten stets redundanzfrei und konsistent gehalten werden müssen. Um dies zu gewährleisten, muss jeder Datensatz eindeutig identifiziert werden können. Zusätzlich dürfen Daten nur einmal in der Datenbank vorhanden sein. Andernfalls wäre keine eindeutige Zuordnung der Daten mehr möglich. Wenn dies zutrifft, liegen die Daten nun in ihrer Normalform vor. Diese Methode wird als Normalisierung bezeichnet (Rausch, 1999).

2.1.4 Objektdatenbanken

Eine Objektdatenbank ist ein weiteres Modell für eine Datenbank. Sie zeichnet sich dadurch aus, dass Daten als Objekte respektive zu Objektorientierung gespeichert und verwaltet werden. Das dazu passende Managementsystem ist das objektorientierte Datenbankmanagementsystem (ODBMS). Ein Objekt kann ein Gegenstand oder ein Begriff sein, der ein oder mehrere objektbeschreibende Attribute trägt. Jedoch enthält das Objekt neben den Attributen noch Daten und Methoden. Methoden sind Funktionen, die verwendet werden, um Objekte und Attribute zu verarbeiten.

Das Objektdatenbankmanagementsystem hat die gleiche Aufgabe wie jedes andere Datenbankmanagementsystem, nämlich die Speicherung von Daten und den zuverlässigen Zugriff auf die gespeicherten Daten. Darüber hinaus gibt es weitere Aufgabebereiche, die erfüllt werden müssen, um als ODBMS zu gelten. Denn im Vergleich zu anderen DBMS muss dieses komplexere, aus verschiedenen Datentypen zusammengesetzte Objekte verwalten und deren Objektidentität sichern. Objektidentität wird gesichert, indem jedem Objekt eine eigene Objekt-ID, die nur einmal vorhanden ist, zugeordnet wird. Objekte sind Objektklassen zugeordnet, die in einer Klassenhierarchie angeordnet sind. Des Weiteren muss ODBMS eine Data Manipulation Language bereitstellen (Gebhardt, 2018).

2.1.5 Dokumentenbasierte Datenbank (NoSQL)

Ein weiterer zu beschreibender Typ einer Datenbank ist die dokumentenbasierte Datenbank. Diese unterscheidet sich von der relationalen und der objektorientierten Datenbank insofern, als sie nicht den relationalen Grundgedanken befolgt. Damit gehört sie zu der NoSQL-Familie, deren Mitglieder aufgrund ihrer horizontalen Skalierung für Big-Data-Anwendungen geeignet sind. NoSQL bedeutet *not only SQL* und ist nicht als grundlegender Verzicht auf SQL zu verstehen. Zwar gibt es Systeme, die vollkommen auf SQL verzichten, doch genauso existieren Systeme, die nur bestimmte Elemente von SQL berücksichtigen. Ein Unterschied besteht darin, dass innerhalb der relationalen Datenbanken mit Spalten und Zeilen gearbeitet wird, während NoSQL-Datenbanken ihre Daten mithilfe von beispielsweise Objekten, Reihen oder Wertepaaren verwalten.

Im Gegensatz zu den relationalen Datenbanken, die dem ACID-Prinzip folgen, liegt NoSQL-Datenbanken das CAP-Theorem zugrunde. Das Akronym CAP steht für [c]onsistency, [a]vailability und [p]artition. Dieses Theorem besagt, dass nur zwei der drei Prinzipien gut umgesetzt werden können. Demzufolge muss eine Entscheidung getroffen werden, welche Eigenschaft für die gegebene Datenbank von Bedeutung ist. Obwohl beide Theorien das Prinzip *Consistency* nennen, definieren sie diese unterschiedlich (Mehra, 2017).

In einer dokumentbasierten Datenbank werden die Daten in Form von Dokumenten, die jeweils eine eigene eindeutige ID besitzen, abgespeichert. Mit der ID ist es möglich, direkten Zugriff auf die Dokumente zu bekommen und diese zu bearbeiten. Da der dokumentbasierten Datenbank kein Schema zugrunde liegt, können nicht strukturierte Daten (bsp. Filme, MPEG) ebenfalls abgespeichert werden. Auch strukturierte Daten, die ein Standarddateiformat haben, können gespeichert werden. In einem Dokument können durch das fehlende Schema

unterschiedliche Daten gelagert werden. Es gibt nur einen Speicher, in dem alle Dokumente liegen und – anders als bei relationalen Datenbanken – findet keine Unterscheidung nach Datentypen statt. Da man Strukturen einer Datei nicht in Tabellen, wie es für relationale Datenbanken der Fall ist, abbilden muss, ist das Abspeichern und Auslesen weniger komplex.

Im Gegensatz dazu können in dokumentbasierten Datenbanken Objekte in ihrer ursprünglichen Form als Dokumente abgespeichert werden. Es gibt zwei gängige Arten, die Daten zu speichern: Zum einen kann das Objekt ohne Modifikationen als Dokument untergebracht werden, zum anderen können die Attribute eines Objekts in ein weiteres Dokument mit Referenz zum Hauptobjekt gekapselt werden. Die zweite Methode wird bei Daten verwendet, die sich fortwährend vermehren oder oft verändern. In Abbildung 1 wird ein Beispiel angeführt, wie der Aufbau eines Dokuments aussehen könnte (Database.Guide, 2016).

```
{
  _id: "1"
  name: "OrderItem 1"
},
{
  _id: "2"
  name: "OrderItem 2"
},
```

Abbildung 1: Struktur von Objekten in einem Dokument

2.2 Objektrelationale Abbildung

Objektrelationale Abbildung ist die Lösung für Object-Relational-Impedance-Mismatch – kurz Impedance-Mismatch (objektrelationaler Bruch). Das Problem taucht auf, wenn Objekte einer objektorientierten Sprache in einer relationalen Datenbank gespeichert werden sollen. Hier muss sich mit dem *strukturellen Bruch* auseinandergesetzt werden, denn bei der Übertragung von Java-Objekten in eine relationale Datenbank gibt es zwei zugrunde liegende Architekturen. Da Objekte aus Attributen, Daten und Methoden bestehen und relationale Datenbanken aus Tabellen mit Spalten und Datensätzen, müssen die Objekte in eine passende Form umgewandelt werden. Dies hat den Nachteil, dass eine weitere Schicht hinzugezogen werden muss. Das Verfahren der Umwandlung der Objekte in RDBMS wird als objektrelationales Mapping (ORM) oder objektrelationales Abbilden von Objekten bezeichnet (IT-Visions.de, 2017).

Im Folgenden wird die Programmiersprache JAVA, mit der die Basissoftware der Firma Köhl entwickelt wurde, untersucht. Andere objektorientierte Programmiersprachen, wie C++, sind für diese Arbeit irrelevant und werden nicht näher beschrieben.

Eine Entität, die persistent in einer Datenbank abgebildet wird, ist ein Plain Old Java Object (POJO). Die Attribute des Objekts werden zu den Spalten in der Tabelle und die Werte der Objektattribute werden zu den Daten der Spalten. Der Name der Tabelle leitet sich von dem Namen der Klasse ab (Abbildung 2). Die Relationen (Beziehungen) zwischen den Objekten werden durch objektrelationale Metadaten ausgedrückt, die in einer XML-Datei abgelegt sind oder indem Java-Annotationen angelegt werden. Diese Assoziationen werden über Fremdschlüssel zwischen den Tabellen abgebildet.



Abbildung 2: Abbildung von Entitäten in eine Tabelle

Durch die Popularität der objektorientierten Programmiersprachen in Kombination mit relationalen Datenbanken tritt das Problem der objektrelationalen Abbildung vermehrt auf. Aus diesem Grund wurden Persistenz-Frameworks geschaffen, die eine direkte Lösung bieten. In der Softwareentwicklung wird von einem Objekt gesprochen, das die Eigenschaft der Persistenz aufweist, wenn dieses permanent und unabhängig vom erzeugten Programm gespeichert werden kann (ITWissen.Info, 2016).

Ein Framework ist eine halb fertige Applikation. Es besteht aus einer allgemeinen und wiederverwendbaren Struktur, die man mit anderen Applikationen teilen kann. Es ist üblich, dass ein Entwickler ein Framework in seine Applikation integriert und diese soweit erweitert, wie es die Anforderungen verlangen (Tahchiev, Leme, & Massol, 2010).

Hier gibt es Spezifikationen wie Java-Persistence-API oder Java Data Objects, die einen Standard bzw. Richtlinien bieten, wie man mit der Problematik des objektrelationalen Bruchs umgeht. Zudem gibt es die eigentliche Implementierung dieser Spezifikation, wie Hibernate oder EclipseLink.

Die Firma Köhl nutzt die Java Persistence API (JPA) als Spezifikation in Kombination mit Hibernate. Aus diesem Grund wird sich diese Arbeit auf diese Programme beschränken. Schnittstellen wie Java Data Objects (JDO), Data Nucleus oder auch die JPA-Referenzimplementierung EclipseLink werden nicht erläutert, da sie für diese Arbeit nicht von Bedeutung sind.

2.2.1 Java-Persistenz-API (JPA)

Java Persistence API (JPA) ist eine Spezifikation, die von der JSR 220 Expert Group erstmals im Mai 2006 im Rahmen eines Projekts veröffentlicht wurde. Aktuell bildet JPA 2.1 die neueste Version, sie wurde 2013 freigegeben (Schmidt, 2013). Diese Spezifikation stellt eine Lösung für den objektrelationalen Bruch dar. Sie beschreibt, wie Objekte einer objektorientierten Applikation in einer relationalen Datenbank persistent gespeichert werden. JPA liefert eine Schnittstelle, jedoch keine Implementierung. In den meisten Fällen werden Frameworks bzw. Provider wie Hibernate oder EclipseLink zur Hilfe genommen.

Diese implementieren die Schnittstelle. JPA kann sowohl mit der Java Standard Edition (Java SE) als auch mit der Java Enterprise Edition (Java EE) genutzt werden. Zudem ist ab Java EE 5 die Spezifikation JPA inkludiert (Horn, 2010). Durch das Einbinden von JPA und deren Implementierung kann sich der Programmierer direkt der Geschäftslogik zuwenden und muss sich nicht mit dem relationalen Bruch auseinandersetzen.

Die Java-Persistence-API gliedert sich neben der Schnittstelle noch in Entity, relationale Metadaten und die Java-Persistence-Query-Language.

Objekte, oder genauer Plain-Old-Java-Objects, sind Entitäten. Diese werden in Tabellen einer relationalen Datenbank persistent abgebildet. Auch ist es mittels der JPA-Schnittstelle möglich, Relationen zwischen den Klassen abzubilden. Abhängigkeiten können die folgenden

sein: One-To-One, One-To-Many, Many-To-One und Many-To-Many. Diese Relationen können uni- oder bidirektional sein. Wenn eine Relation bidirektional ist, heißt das, dass in beiden Klassen eine Referenz auf die jeweils andere Klasse existiert (Lars Vogel, 2017).

Die Abbildung erfolgt mittels objektrelationaler Metadaten, die in Form von XML-Dateien⁷ oder Java-Annotationen vorliegen. Annotationen in Java werden mit einem @, gefolgt von einem Bedeutung tragendem Wort gebildet. Anhand des Wortes und dem @-Zeichen wird erkannt, welche Funktion ausgelöst werden soll (Horn, Torsten-Horn.de, 2008).

XML-Daten sollten zur Erstellung von Projekten genutzt werden, da sie die Konfiguration von Projekten ermöglichen, ohne deren Kompilierung zu erfordern.

Wie bereits erwähnt, bietet JPA die Datenbanksprache Java-Persistence-Query-Language (JPQL) an. Sie ist ein SQL-Dialekt, wobei die Abfragen nicht auf Datenbanktabellen abzielen, sondern auf gespeicherte Entitäten. Abfragen, die mit JPQL formuliert wurden, werden durch JPA in ein SQL-Statement transformiert und an die Datenbank gesendet. Da man in der Regel nur JPQL-Abfragen formuliert, die durch JPA in SQL übersetzt werden, ist es möglich, die darunter liegende Datenbank zu wechseln, ohne Anpassungen an den Java-Klassen zu vollziehen. Zusätzlich ist es möglich, Abfragen in SQL zu formulieren, hierbei spricht man von einer Native SQL. Solche Queries werden jedoch nicht von JPA kontrolliert und der Nutzer muss selbst drauf achten, dass die Zieldatenbanken die Query verstehen (Juneau, 2018).

2.2.2 Hibernate

Hibernate ist die Implementierung der Java-Persistence-API und übernimmt sowohl das Abbilden von Objekten in eine relationale Datenbank als auch das Rekonstruieren in ein Objekt. Hibernate ist ein von JBoss, Inc., (Red Hat) entwickeltes Open-Source-Framework, das unter der Open Source GNU Lesser General Public License (LGPL) zugelassen ist (Tutorialspoint, 2019). Die aktuellste Version ist Hibernate 5.3.0.Final, die für die objektorientierte Programmiersprache Java entwickelt wurde und im Mai 2018 erschien (Wikipedia, 2019).

Hibernate arbeitet in der Sprache Hibernate Query Language (HQL), mit der man Abfragen an persistent gespeicherte Objekte formulieren kann. Sie verbindet im Grunde die relationalen Datenbanken mit der objektorientierten Programmierung. HQL ähnelt JPQL in vielen Punkten und basiert auf SQL. Des Weiteren wird HQL durch die Schnittstelle Java Database Connectivity in einen SQL-Dialekt umgewandelt. Welcher Dialekt ausgewählt wird, entscheidet die Datenbank, mit der gearbeitet wird (Haß, 2019). Abbildung 3 zeigt ein einfaches HQL-Abfragebeispiel, das alle Objekte der Tabelle Orders abfragt.

`SELECT o FROM Orders o`

Abbildung 3: Abfragebeispiel in Hibernate Query Language

Objektrelationale Abbildungen erfolgen mittels einer XML-Datei oder über Java-Annotationen. Annotationen sind Sprachelemente aus der Java-Programmiersprache. Mit diesen kann man Metadaten in die Programmierung einfließen lassen (Wikipedia, 2019). Sie können entweder Aussagen über Methoden enthalten, wie `@Column`, oder eventuell Aussagen über den Verfasser eines Codeausschnitts (`@Author`). Tabelle 4 zeigt einige Beispiele von Java-Annotationen und deren Bedeutung für Hibernate.

⁷ XML: Extensible Mark-up Language (Augsten, 2018)

Annotation	Modifier	Beschreibung
@Entity		Durch diese Markierung wird die Klasse als Hibernate-Entität angesehen.
@Table	Name	Ordnet diese Klasse einer Datenbanktabelle zu, die durch den Namensmodifikator angegeben wird. Wenn der Name nicht angegeben wird, ordnet er die Klasse einer Tabelle zu, die denselben Namen wie die Klasse hat.
@Id		Markiert dieses Klassenfeld als Primärschlüsselspalte.
@GeneratedValue		Weist die Datenbank an, automatisch einen Wert für dieses Feld zu generieren.
@Column	Name	Ordnet dieses Feld durch den angegebenen Namen der Tabellenspalte zu und verwendet den Feldnamen, wenn der Namensmodifikator fehlt.

Tabelle 4: Beispiele für Annotationen (Yatin, 2017)

Es gibt drei Arten, um Vererbungsbeziehungen zu modellieren und mit Hibernate ist es möglich, sie alle abzubilden. Zum einen kann für jede Vererbungshierarchie eine Tabelle erstellt werden und, wenn nötig, für jede Unterklasse eine weitere Tabelle oder eine Tabelle für eine konkrete Klasse zur Verfügung stellen (Horn, torsten-horn.de, 2007).

Grundbefehle, um auf die Datenbank zuzugreifen, sind von Hibernate vorgegeben. Mit den Basis-Operatoren von Hibernate lassen sich Entitäten in der Datenbank durch CRUD-(Create, Read, Update und Delete-)Operationen verwalten, die die Schnittstelle zwischen Benutzer und Datenbank bilden (Yatin, 2017)).

In Tabelle 5 befindet sich eine Auflistung der CRUD-Befehle, ihrer Bedeutung, wie diese in SQL realisiert und im Java-Code geschrieben werden.

CRUD	Bedeutung	SQL	Java
Create	Neuer Datensatz anlegen	Insert	Save(Object)
Read	Daten abfragen	Select	Get(Object)
Update	Datensatz updaten	Update	Set()
Delete	Datensatz löschen	Delete	Delete(Object)

Tabelle 5: CRUD-Operationen und deren Umsetzung in SQL und Java

Die Vorteile, diese Befehle aus der Applikation aufzurufen, bestehen in der Datensicherheit und dem Datenzugriff, denn es werden ausschließlich Query- bzw. Hostsprachen benutzt. Darüber hinaus wird die Datenintegrität erhöht und die Unabhängigkeit von Drittapplikationen gewährleistet (Yatin, 2017).

2.2.3 Anwendungsbeispiel Hibernate

Im folgenden Kapitel werden drei Hauptkomponenten von Hibernate in Form eines Anwendungsbeispiels erläutert. Die erste zu erläuternde Komponente ist die Konfigurationsdatei, die Basiseinstellungen zum Datenbankzugriff beinhaltet. Weiter wird darauf eingegangen, wie man aus Klassen Entitäten macht und schließlich werden Data Access Objects, mit denen man auf Entitäten zugreift, erläutert.

Um mit Hibernate und der Datenbank zu arbeiten, muss eine Kommunikation mit der Datenbank aufgebaut werden. Dies geschieht in der Hibernate-Konfigurationsdatei *persistence.xml*. In Abbildung 4 ist mit dem Tag `<jta-data-source>` eingeleitet, dass die Daten, die verwendet wurden, wie Name der Datenbank, Benutzer und Passwort, ausgelagert wurden und mit

`java:/koehl` eine Referenz zwischen beiden Dateien erstellt wird. Spezielle Eigenschaften werden mit *property*, gefolgt von einem *name*-Attribut, dem mit dem Gleichheitszeichen eine Bedeutung zugewiesen wird, erstellt. Das *name*-Attribut hat den Präfix *hibernate*, da als Provider Hibernate verwendet wurde.

```
<persistence
  xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="koehl">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/koehl</jta-data-source>
    <properties>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.dialect"
value="org.hibernate.dialect.SQLServerDialect" />
      <property name="hibernate.hbm2ddl.auto"
value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Abbildung 4: Auszug aus der Hibernate-Datei `persistence.xml`

Durch *hibernate.show_sql* kann mit dem Parameter *true* der von Hibernate generierte SQL-Code, mit dem das Schema erstellt wurde, angezeigt werden. Die Eigenschaft *hbm2ddl.auto* exportiert ein Schema zur Datenbank, wenn die *SessionFactory* erstellt wird. Tabelle 6 zeigt von *hbm2ddl.auto* verwendete Parameter und deren Bedeutung (Hibernate Community Documentation, n.d.).

Hbm2ddl.auto	Bedeutung
validate	Überprüft das Schema auf seine Korrektheit hin. Verändert die Datenbank nicht
update	Schema wird aktualisiert
create	Erstellt ein Schema und löscht vorhandene Daten
create-drop	Löscht das Schema explizit, wenn die Session abläuft

Tabelle 6: Parameter `Hibernate.hbm2ddl.auto` (Hibernate Community Documentation, n.d.)

Mit der Konfigurationsdatei (Abbildung 4) eingestellt, wird der *EntityManager* zur Hilfe genommen, um den Austausch zwischen der Applikation und der Datenbank zu ermöglichen. Abbildung 5 zeigt ein Fragment dessen, wie ein Objekt vom *EntityManagerFactory* mit dem Parameter der XML Konfigurationsdatei, erzeugt wird. Mit der Factory werden *EntityManager*-Instanzen erzeugt, diese können Datenbankabfragen ausführen. Durch eine Instanz von der Klasse *Query* kann man einen *EntityManager* eine HQL-Anfrage ausführen lassen.

```

EntityManagerFactory emf = Hibernate ... createEntityManagerFactory(hibernate.cfg.xml)
EntityManager em = emf.createEntityManager();
Query q = em.createQuery („select o from orders o“)

```

Abbildung 5: Erzeugung eines EntityManagers mittels der EntityManagerFactory

Abbildung 6 zeigt eine Klasse, die in eine Datenbank gemappt wird. Diese Klasse enthält zwei Attribute und diverse Methoden. Um eine Instanz dieser Klasse in eine Datenbank zu speichern, muss diese Klasse mit der `@Entity`-Annotation versehen sein. Damit wird diese Klasse für Hibernate zu einer Entität, während Hibernate die relationale Abbildung übernimmt und eine Instanz davon in die Datenbank speichert. Zusätzlich muss diese Entität eindeutig zu identifizieren sein, wodurch das Attribut `customerID` mit der Annotation `@id` zum Schlüsselattribut wird. Mit der Annotation `@Column` über der `getCustomerID()`-Methode wird der Name dieser Spalte in der Datenbank benannt.

```

@Entity
@Table(name = "Customer")
public class Customer implements Serializable {
    private int customerID;
    private String firstName;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "CUSTOMER_ID")
    public int getCustomerID() {
        return customerID;
    } //getter und setter
}

```

Abbildung 6: Klasse Customer.java mit Annotationen

Attribute besitzen Datentypen, die von Hibernate erkannt und richtig abgebildet werden. Jedoch gibt es Datentypen, die mehrdeutig sind, wie z. B. in der Programmiersprache Java der Datentyp `Date`. Aus diesem Grund muss ein mehrdeutiges Attribut mit einer Annotation verknüpft werden, die die Art der Interpretation innerhalb der relationalen Datenbank angibt. Mit der Annotation `@Temporal` (`TemporalType.Date`) wird beispielsweise angegeben, dass es sich um ein Datum mit Tag, Monat und Jahr handeln soll. Komplexe Datentypen werden mittels Assoziationen umgesetzt. Hier wird die Annotation `@OneToMany`, `@ManyToOne` und `@ManyToMany` verwendet. In der Abbildung 7 wird eine One-to-Many-Relation dargestellt. Diese Klasse enthält ein `Set`-Attribut, das Order-Item-Objekte enthalten kann. Zusätzlich wird in der `getOrderItems()`-Methode die Annotation `@OneToMany` verwendet, um Hibernate die Art der Beziehung mitzugeben (Hibernate Community Documentation, n.d.).

```

private Set<OrderItem> orderitems = new HashSet<OrderItem>();

@OneToMany(mappedBy = "primaryKey.product", cascade =
CascadeType.ALL)
public Set<OrderItem> getOrderItems() {
    return orderitems;
}

```

Abbildung 7: Codeausschnitt einer Klasse mit einer One-to-Many Beziehung

Der *EntityManager* enthält Methoden, um mit der Datenbank zu kommunizieren. Tabelle 7 zeigt die Methoden, die vom *EntityManager* vorgegeben werden, und deren Bedeutung.

Nr.	Methode EntityManager	Bedeutung
1	Persist(o: Object)	Objekt in DB speichern
2	Find(c: Class, id: String)	Bestimmte Entity aus DB zurückliefern lassen
3	Refresh(o: Object)	Entität aktualisieren
4	Remove(o: Object)	Entität entfernen
5	CreateQuery(query: String)	HQL-Query erzeugen
6	GetTransaction(): EntityTransaction	Verbindung aufbauen

Tabelle 7: Methoden des EntityManagers

Im Gegensatz zu einer Java-Methode, die in der Regel Attribute und Methoden beinhaltet, werden unter Hibernate die Methoden in separaten Klassen gekapselt. Diese Klassen nennt man Data Access Objects (DAO). Grundsätzlich wird für jede Entitätenklasse eine DAO-Klasse erstellt. Diese beinhaltet alle wichtigen Methoden, die die Arbeit mit den Objekten ermöglichen, und zwar unter Verwendung der in Tabelle 7 genannten Methoden, jedoch müssen diese vom Benutzer selbst geschrieben werden.

Die Nutzung eines DAO erfolgt, indem zunächst ein Objekt von DAO erzeugt wird, um auf die Funktionalität der Methoden zuzugreifen. Anschließend wird ein Objekt von der Entität erzeugt, das den Rückgabewert, beispielsweise aus einer Getter-Methode, vom DAO-Objekt zugewiesen bekommt. Im Hintergrund wird durch den *EntityManager* eine Verbindung zur Datenbank eröffnet, eine Transaktion begonnen, Methoden ausgeführt und schlussendlich mit einem *commit()*, gefolgt von dem Return-Wert, die Transaktion beendet (Bauer & King, 2007).

2.3 Partitionierung

Die Datenbankpartitionierung ist eine in SQL Server verfügbare Funktion, in der Tabellen und Indizes horizontal in Einheiten, genauer gesagt Partitionen, unterteilt werden. Horizontal in dem Sinne, dass eine Menge von Datensätzen in einer Partition abgebildet wird. Diese Einheiten lassen sich in eine oder mehrere Dateigruppen (Filegroups) innerhalb einer Datenbank verteilen. Die Filegroups können auf unterschiedlichen Speichermedien gespeichert werden. Zu beachten ist, dass alle Partitionen eines Index oder einer Tabelle auf derselben Datenbank liegen müssen.

Mit dem SQL Server 2005 wurde diese Funktion veröffentlicht, jedoch war diese nur in der Enterprise-, der Entwickler- und der Evaluation-Edition erhältlich (**Microsoft SQL Dokumentation, 2016**). Mit dem Erscheinen des SQL Servers 2016 (13.x) SP1 wurde diese Funktion integriert und für alle folgenden Versionen zur Verfügung gestellt.

Vorteil der Partitionierung ist zum einen, dass die Übertragung von Daten bzw. der Zugriff auf eine Teilmenge von Daten effizient verläuft und ohne die Datenintegrität zu gefährden. Durch die Einteilung der Daten in kleinere Einheiten werden zusätzlich Wartungsvorgänge zügiger abgearbeitet, da diese Operationen nur auf einer Teilmenge der Daten ausgeführt werden (z. B. das Komprimieren von Daten und Back-ups). Zudem können Query-Abfragen durch den Query-Optimizer hinsichtlich ihrer Performanz erhöht werden. Beispielsweise können *join*-Abfragen zwischen zwei oder mehreren partitionierten Tabellen schneller verarbeitet werden. Ebenfalls findet die Partition-Eliminierung bei Abfragen, die bestimmte Kriterien enthalten, statt. So werden ausschließlich relevante Partitionen bei der Abfrage berücksichtigt und irrelevante werden außer Acht gelassen.

Die Partitionierung ist keine Archivierungsmethode (siehe Kap. 4), sondern kann in die Kategorie der logischen Archivierung eingegliedert werden. Jedoch kann diese Methode Abhilfe schaffen, wenn Datensätze aus einer Partition in eine Hilfstabelle überführt und anschließend archiviert werden sollen. Partitionierung ist insofern vorteilhaft, als dass die Überführung der partitionierten Daten schnell vonstattengeht (**Microsoft SQL Dokumentation, 2016**).

Im Folgenden werden die für die Partitionierung notwendigen Komponenten, die Partitionsfunktion und das Partitionsschema, erläutert und in Form eines Anwendungsbeispiels veranschaulicht.

2.3.1 Partitionsfunktion

Die Partitionsfunktion gibt an, wie Datensätze oder ein Index in Partitionen abgebildet werden. Sie basiert auf Werten (Value), die angegeben werden müssen und die Grenzen einer Partition festlegen. Bei der Erstellung der Grenzen kann man zwischen *Range LEFT* und *Range RIGHT* entscheiden. Dies gibt an, wie ein Datensatz sich verhalten soll, wenn dessen Kriterium genau auf eine Grenze fällt. Im Falle von *LEFT* würde der Datensatz in die Partition fallen, die kleiner ist als die gesetzte Grenze, und bei *RIGHT* würde der Datensatz in die Partition fallen, die größer ist als die gesetzte Grenze. In Tabelle 8 werden die Grenzen mit den *Integer*-Werten 10 und 100 besetzt. Ein Datensatz, der das Kriterium 10 beinhaltet, würde im Falle von *LEFT* in der Partition 1 abgebildet werden und bei *RIGHT* in der zweiten Partition.

Partitionen	1	2	3
Left for values	$C1 \leq 10$	$C1 > 10 \text{ AND } C1 \leq 100$	$C1 > 100$
Right for values	$C1 < 10$	$C1 \geq 10 \text{ AND } C1 < 100$	$C1 \geq 100$

Tabelle 8: Verhalten der Datensätze bei Range LEFT und Range RIGHT

Zudem nimmt diese Funktion einen Parameter entgegen, dessen Datentyp eine Spalte ist und der gewählt wurde, um eine Tabelle zu partitionieren. Alle Spalten, die geeignet wären, ein Schlüssel einer Tabelle (Date, Integer) zu sein, eignen sich auch als Partitionsspalte. Abbildung 8 zeigt die Partitionsfunktion mit dem *INT*-Datentyp als Parameter, den Werten der Grenzen 10 und 100 für die Partitionen und den *LEFT*-Operator (Retama, 2011).

```
Create Partition Function myPartitionFunction(INT) AS
Range LEFT FOR VALUES (10,100)
```

Abbildung 8: Partitionsfunktion

2.3.2 Partitionsschema

Das Partitionsschema spezifiziert, in welche Filegroups die von der Partitionsfunktion gebildeten Partitionen abgelegt werden sollen. Die in Filegroups gelagerten Dateien nennt man Secondary-Files (.ndf). In diese Dateien können Datensätze gespeichert werden. Secondary-Files werden so zugewiesen wie Dateien in einem Ordner in einem System. Es ist möglich, mehrere Dateien in eine Filegroup zu legen. In diesem Fall würden alle Datensätze unter den Secondary-Files in dieser Filegroup verteilt.

Es wird der Partitionsfunktion aus Abbildung 8 vermittelt, dass der Inhalt der erstellten Partitionen in die Filegroups FG1, FG2 und FG3 abgelegt werden soll. Das Erstellen der Filegroups ist optional und dient dazu, die Dateien im späteren Verlauf auf unterschiedliche Datenträger

zu speichern. Jedoch ist es auch möglich, alle Dateien in die Filegroup, beispielsweise Primary, die standardmäßig bereits erstellt wird, zu setzen. In dem Fall, dass die Partitionsfunktion zwei Grenzen betitelt, werden im Endeffekt drei Partitionen gebildet. Mit der in Abbildung 9 gezeigten Partitionsfunktion würde der SQL Server Partition 1 der FG1 zuordnen, Partition 2 der FG2 und Partition 3 der FG3 (Retama, 2011).

```
Create Partition Scheme myPartitionScheme AS
Partition myPartitionFunction to (FG1, FG2, FG3)
```

Abbildung 9: Partitionsschema

2.3.3 Indexierung und Datenausrichtung

Es gibt mehrere Möglichkeiten, eine Tabelle mit Indizes zu versehen und diese zu partitionieren. Die beste Methode ist, den Index ebenfalls so zu partitionieren wie die einzelnen Partitionen. Hier wird von einem ausgerichteten Index (Aligned-Index) gesprochen. In so einem Fall werden die Indexteile in die gleiche Datei (.ndf) wie die zugehörige Partition gespeichert. Dies ermöglicht es, Funktionen zu nutzen, die als Ziel vereinzelte Partitionen haben. Somit ist es möglich, von einer Partition ein Back-up zu erstellen bzw. einzuspielen. Ebenfalls werden die Abfragezeiten der Queries performanter, da der Query-Optimizer nur den Index in Betracht zieht, der zur jeweiligen Partition zugehörig ist. Ebenso kann auf diese Art der komplette Inhalt einer Partition effizient, also nicht durch das Verschieben der einzelnen Datensätze, sondern nur durch die Nutzung von Metadaten, in eine andere Tabelle verschoben werden.

Dies wird ermöglicht, indem aus dem eigentlichen primären Schlüssel ein zusammengesetzter Schlüssel kreiert wird, der sich aus der Spalte mit dem primären Schlüssel und der Partitionsspalte zusammensetzt.

Abbildung 10 zeigt die Erzeugung einer Tabelle mit Attributen und dem zusammengesetzten gruppierten Schlüssel. Durch die erste *ON*-Klausel, wo der primäre Schlüssel *PK_myTable* erzeugt wird, wird dieser auch dem Partitionsschema zugeordnet, um den Index an den Daten auszurichten. In der zweiten *ON*-Klausel wird die erstellte Tabelle in das Partitionsschema *myPartitionScheme* gesetzt und die Spalte *col2* als Partitionsspalte mitgegeben. Demzufolge werden jeder Datensatz sowie der Index, der in diese Tabelle gespeichert werden soll, durch die Partitionsfunktion geprüft und durch das Partitionsschema in die entsprechende Filegroup gespeichert (Retama, 2011).

```
Create Table myTable
(
    col1 int identity NOT NULL,
    col2 datetime NOT NULL,
    col3 varchar(20),
    col4 varchar(20),
    Constraint PK_myTable primary key clustered (col1, col2)
    ON myPartitionScheme(col2)
) on myPartitionScheme(Col2)
go
```

Abbildung 10: Erstellung einer Tabelle unter Anwendung des Partitionsschemas

2.3.4 Partitionsfunktionen

Die *Split*-Funktion ist die erste Partitionsfunktion, die in diesem Kapitel behandelt wird. Diese Funktion ermöglicht es durch das Hinzufügen einer weiteren Grenze, weitere Partitionen zu erstellen. Die Datenbank-Engine erzeugt eine weitere Grenze, die den vorhandenen Bereich in zwei Bereiche aufteilt und somit eine weitere Partition schafft. Die bereits vorhandenen Daten werden der entsprechenden Partition zugeordnet. Üblich ist es, die letzte Partition frei von Daten zu halten. Mit der *Split*-Funktion wird eine weitere Grenze erzeugt, jedoch muss das Datenbankmanagementsystem nicht beachten, wie die Daten welcher Partition zugeordnet werden, da in der neusten Partition keine Daten vorhanden sind. Weil das Datenbankmanagementsystem sich nicht mit der Zuteilung der Daten beschäftigen muss, erfolgt das Erstellen einer neuen Partition zeiteffizient. Abbildung 11 zeigt die SQL Server-Syntax der *Split*-Funktion (Retama, 2011).

```
Alter Partition Function myPartitionFunction()  
Split Range (boundary value)
```

Abbildung 11: Partitionsfunktion – Split

Im Partitionsschema wurde definiert, welche Dateigruppen (Filegruppen) genutzt werden sollen. Im Fall, dass eine *Split*-Funktion angewandt wurde und eine weitere Partition existiert, muss das dem Partitionsschema mitgeteilt werden. Wenn Datensätze in der neuen Partition in eine neue Filegruppe gespeichert werden sollen, muss dafür zunächst eine neue Gruppe erstellt werden. Dies kann über das Microsoft SQL Server Management Studio erfolgen, indem die Eigenschaften einer Datenbank aufgerufen werden und mit der Schaltfläche *Hinzufügen* weitere Filegroups gefertigt werden. Weitere Gruppen können via Quellcode erstellt werden, dafür wird, wie in Abbildung 12 dargestellt, der Datenbank mit dem *Add*-Operator eine neue Filegruppe zugewiesen, ein neuer File (Datei) mit den Parametern *Name*, *Speicherort*, *Größe*, *maximale Größe* erschaffen und diese Datei der Filegruppe zugeordnet (Retama, 2011).

```
Alter Database DatabaseName ADD Filegroup FilegroupName  
Alter Database DatabaseName ADD File  
(  
    Name = P4,  
    FileName = 'storage path',  
    size = 1204KB, MaxSize = UNLIMITED, Filegrowth = 10%  
) to FILEGROUP FilegroupName
```

Abbildung 12: Filegroup via Quellcode erstellen

Anschließend muss vorher das Partitionsschema mit dem Befehl, wie in Abbildung 13 zu sehen, so angepasst werden, dass es die neue Filegroup beinhaltet, damit die Partitionsfunktion die neuen Daten in diese abbilden kann.

```
Alter Partition Scheme myPartitionScheme NEXT USED [FilegroupName]
```

Abbildung 13: Alter Partition Scheme Next Used

Um Partitionen miteinander zu verschmelzen, wird der *Merge*-Operator verwendet. Die Syntax, die in Abbildung 14 zu sehen ist, ähnelt der *Split*-Funktion. Die Funktion nimmt einen Parameter entgegen, der in der Partitionsfunktion als Grenze definiert worden ist. Wenn die Funktion ausgeführt wird, sorgt diese Funktion dafür, dass die als Parameter angegebene Grenze aus der Partitionsfunktion entfernt wird. Dies hat zur Folge, dass die Partition mit Werten kleiner als

die entfernte Grenze und die Partition mit Werten größer als die Grenze zu einer verschmelzen und aus den Datensätzen beider Partitionen bestehen.

Würde jede Partition in eine separate Filegroup geschrieben werden, würde sich diese Kombination aus beiden Partitionen in der aktuellen Filegroup wiederfinden bzw. der mit der nächst höheren Grenze. Zu beachten ist, dass man nur eine Grenze auf einmal auflösen kann. Damit mehrere Partitionen miteinander verbunden werden, müsste dieser Schritt wiederholt werden (Retama, 2011).

```
Alter Partition Function myPartitionFunction()  
Merge Range (boundary value)
```

Abbildung 14: Partitionsfunktion – Merge

Mit dem *Switch*-Operator ist es möglich, Daten einer Tabelle performant in eine andere Tabelle zu überführen. Es gibt drei Anwendungsmöglichkeiten für den *Switch*-Operator. Zum einen ist es möglich, alle Daten einer nicht partitionierten Tabelle in eine Partition einer partitionierten Tabelle zu übertragen. Zweitens kann eine Partition einer partitionierten Tabelle in eine andere Partition einer partitionierten Tabelle überführt werden. Die dritte Möglichkeit besteht darin, Partitionen einer partitionierten Tabelle in eine bereits vorhandene Tabelle, die nicht partitioniert ist, zu verschieben.

Abbildung 15 zeigt die Syntax der *Switch*-Funktion. In eckigen Klammern ist *Partition x* und *Partition y* angegeben, wobei *Partition x* die Quellpartition und *Partition y* die Zielpartition ist und x, y die Bezeichnung der Partitionen repräsentieren. Im Fall, dass Daten aus einer Partition in eine andere Partition geschoben werden sollen, müssen beide Partitionen angegeben werden. Wenn Daten aus einer nicht partitionierten Tabelle in eine partitionierte Tabelle verlagert werden sollen, ist die Angabe von *Partition y* erforderlich. Wenn wiederum Daten aus einer partitionierten Tabelle in eine nicht partitionierte Tabelle überführt werden sollen, wird die Angabe von *Partition x* notwendig (Retama, 2011).

Zu beachten ist, dass eine Verschiebung von Daten nur stattfinden kann, wenn die Zieltabelle bzw. die Zielpartition vorhanden und leer ist. Zusätzlich müssen bei der Übertragung von Daten Quelle und Ziel, Tabelle oder Partition in derselben Filegruppe existent sein. Ebenso müssen die Indizes bzw. die Teilindizes in der gleichen Gruppe gegeben sein, und Quelle und Ziel dürfen nicht identisch sein (Microsoft SQL Dokumentation, 2019)

```
Alter Table sourceTable Switch [Partition x]  
to targetTable [Partition y]
```

Abbildung 15: Partitionsfunktion – Switch

2.3.5 Anwendungsbeispiel

In diesem Kapitel werden die grundlegenden Funktionen der Partitionierung anhand eines funktionierenden Beispiels an einer *Order*-Tabelle angewendet und näher erläutert. Zuerst wird eine Partitionsfunktion, dann das Partitionsschema und im Anschluss eine Tabelle mit deren Schlüssel und Indizes erzeugt.

Hauptbestandteil wird es jedoch sein, die vorgestellten Partitionsfunktionen anhand eines Beispiels darzustellen.

Mit dem SQL Server Management Studio lassen sich Filegroups einer Datenbank hinzufügen, indem man deren *Eigenschaften* aufruft (siehe Abbildung 16). Im zweiten Schritt navigiert man zu dem Tab-Dateien und ergänzt neue Dateigruppen (Filegroups) mit der

Schaltfläche *Hinzufügen*. Der Name der Partition wie auch der Name der Filegruppe ist für den Benutzer frei wählbar.

Es wurden drei Dateigruppen (FG1, FG2, FG3) erzeugt und mit logischen Namen (P1, P2, P3) betitelt. In Abbildung 16 sind die Datenbankeigenschaften der Datenbank *GrundlagenPartitionierung* zu sehen.

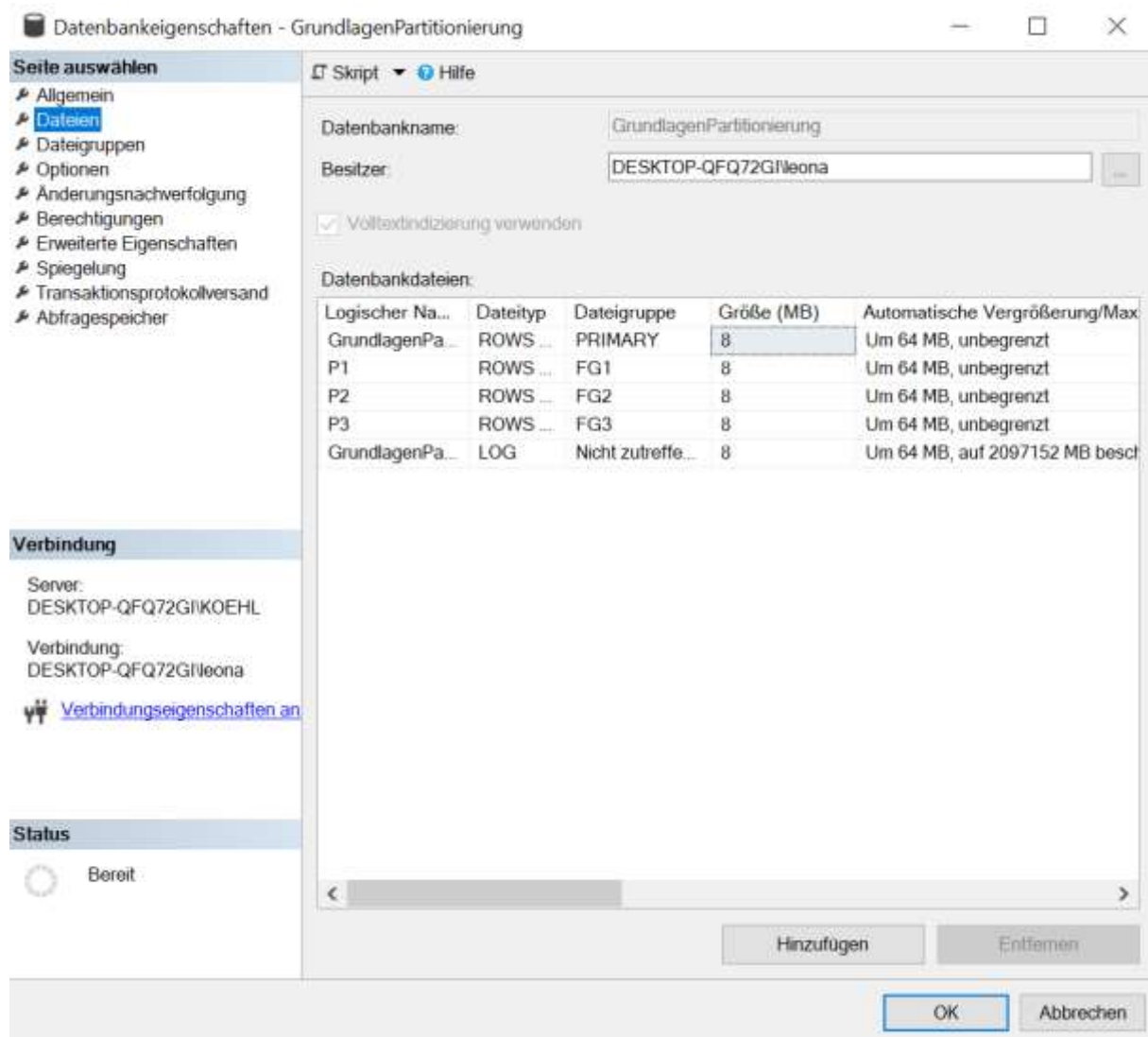


Abbildung 16: Filegroups (FG1, FG2, FG3) erstellen

Folglich, wie in Abbildung 17 zu sehen, werden die Partitionsfunktion *myPartitionFunction* mit dem Datentyp *datetime*, dem *RIGHT*-Operator und den Grenzen *'2003/01/01'* sowie *'2004/01/01'* erzeugt. Anschließend wird das Partitionsschema *myPartitionScheme* erstellt, das die von der *myPartitionFunction* hergestellten Partitionen in die in Abbildung 16 erzeugten Filegruppen delegiert. Anschließend wird die Tabelle *myOrderTable* mit dem gruppierten Index erstellt und an den Daten ausgerichtet und schlussendlich die Tabelle dem Partitionsschema hinzugefügt.

```
--Partitionsfunktion erstellen
Create Partition Function myPartitionFunction(datetime) AS
Range RIGHT FOR VALUES ('2003/01/01', '2004/01/01')
go

--Partitionsschema erstellen
Create Partition Scheme myPartitionScheme AS
Partition myPartitionFunction to (FG1, FG2, FG3)
go

--Tabelle erstellen
Create Table myOrderTable
(
    OrderID int identity NOT NULL,
    OrderDate datetime NOT NULL,
    Amount int,
    ProduktID int,
    Constraint PK_myOrderTable primary key clustered (OrderID,
OrderDate)
ON myPartitionScheme (OrderDate)
) on myPartitionScheme (OrderDate)
go
```

Abbildung 17: Partitionsfunktion und Partitionsschema angewendet

Nun wird die erstellte Tabelle *myOrderTable* mit dem in Abbildung 18 dargestellten Code mit Datensätzen befüllt. Der Code zeigt eine *While*-Schleife, die 300-mal ein Insert in die Datenbank tätigt, wobei in jedem Durchlauf die Attribute mithilfe der Laufvariable *@i* hochgezählt werden. Dieser Code wird zweimal hintereinander durchlaufen, während beim zweiten Durchlauf die Jahreszahl *@OrderDate* um 1 erhöht wird.

```
declare @OrderDate Datetime
Declare @i int
set @OrderDate = '2002/01/01' --R1 ('2002/01/01'), R2 ('2003/01/01'
set @i = 0

while @i < 300
Begin
    insert myOrderTable (OrderDate, Amount, ProduktID)
    values (@OrderDate + @i, @i + 5, @i)
    set @i = @i + 1
end
go
```

Abbildung 18: Tabelle mit Datensätzen befüllen

Im SQL Server Management Studio wird in den Eigenschaften der Tabelle unter dem Reiter *Speicher* angezeigt, ob diese Tabelle partitioniert ist und wie viele Partitionen diese enthält. Zusätzlich werden Partitionsspalte und Partitionsschema angezeigt. Um eine genauere Darstellung zu bekommen, welche Partitionen angelegt worden sind und wie viele Datensätze enthalten sind, kann der Code aus Abbildung 19 angewandt werden, der die Systemvariable *\$Partition* nutzt.

```

select $partition.mypartitionfunction (m.OrderDate) as
Partitionsnummer
, min (m.OrderDate) as [Min Value]
, max (m.OrderDate) as [Max Value]
, count (*) [Datensätze in Partition]
from myOrderTable m
Group by $partition.mypartitionfunction (m.OrderDate)
Order by Partitionsnummer

```

Ergebnisse		Meldungen		
	Partitionsnummer	Min Value	Max Value	Datensätze in Partition
1	1	2002-01-01 00:00:00.000	2002-10-27 00:00:00.000	300
2	2	2003-01-01 00:00:00.000	2003-10-27 00:00:00.000	300

Abbildung 19: Ansicht mit Informationen einer Partition (Doward, 2019)

Um eine bessere Ausgabe zu erhalten, wird in der weiteren Arbeit die View *PartitionInfo* aus Abbildung 20 genutzt (Randal, Tripp, & Delaney, 2009). Dadurch ergibt sich ein kompletter Überblick über die Informationen der Partitionierung, was zur vereinfachten Nutzung führt

Abbildung 21 zeigt die Rückgabe der erstellten Viw. Zu sehen ist, dass die Daten für das Jahr 2002 in die erste Partition und damit auch in die Filegroup FG1 gespeichert wurden, während die Daten von 2003 sich in der zweiten Partition befinden. Partition Nummer 3 und somit auch Filegroup FG3 sind noch leer und haben keine Obergrenze gesetzt (siehe Abb.21, Spalte Value).

Im Vergleich zur Abbildung 19 sind zusätzliche Spalten wie der Objektname, die Filegroup, Total Pages und Comparison (Vergleich) zu sehen.

```

Create View PartitionInfo AS
select OBJECT_NAME (i.object_id) as OBJEKT_NAME,
p.partition_number, fg.NAME AS FILEGROUP_NAME, ROWS, au.total_pages,
case boundary_value_on_right
WHEN 1 THEN 'less than'
ELSE 'Less or equal than' END AS 'Comparition', VALUE
FROM sys.partitions p
JOIN sys.indexes i ON p.object_id = i.object_id
AND p.index_id = i.index_id
JOIN sys.partition_schemes ps ON ps.data_space_id = i.data_space_id
JOIN sys.partition_functions f ON f.function_id = ps.function_id
LEFT JOIN sys.partition_range_values rv
ON f.function_id = rv.function_id
AND p.partition_number = rv.boundary_id
JOIN sys.destination_data_spaces dds
ON dds.partition_scheme_id = ps.data_space_id
AND dds.destination_id = p.partition_number
JOIN sys.filegroups fg ON dds.data_space_id = fg.data_space_id
JOIN (SELECT container_id, SUM(total_pages) as total_pages
FROM sys.allocation_units
Group by container_id) as au
ON au.container_id = p.partition_id
Where i.index_id < 2
Go

```

```
Select * from PartitionInfo
```

Abbildung 20: Erstellung der View PartitionInfo (Randal, Tripp, & Delaney, 2009)

Ergebnisse		Meldungen					
	Objekt_Name	partition_number	filegroup_name	rows	total_pages	Comparition	VALUE
1	myOrderTable	1	FG1	300	9	less than	2003-01-01 00:00:00.000
2	myOrderTable	2	FG2	300	9	less than	2004-01-01 00:00:00.000
3	myOrderTable	3	FG3	0	0	less than	NULL

Abbildung 21: Ausgabe der View PartitionInfo

Da alle Teile der Partition erstellt wurden, werden nun die Partitionsfunktionen angewendet. Als erstes wird mit der *Split*-Funktion eine weitere Partition für die Daten bis '2005/01/01' erzeugt (Abb. 22). Um dies zu tun, muss jedoch zuerst eine neue Filegroup (FG4) erstellt werden, die den Namen P4 bekommt. Die Erzeugung einer weiteren Filegroup erfolgt wie in Abbildung 16 aufgeführt. Anschließend kann die erstellte Filegroup, wie in Abbildung 22 dargestellt, dem Partitionsschema zugeordnet werden (Retama, 2011).

```

Alter Partition SCHEME myPartitionScheme NEXT USED FG4
Alter Partition Function myPartitionFunction()
Split Range ('2005/01/01')

```

Abbildung 22: Partition Next Used und Split-Partition

Damit wurde eine weitere Obergrenze erzeugt und Filegroup FG4 dem Partitionsschema hinzugefügt. Um zu prüfen, ob die Methode wie gewünscht funktioniert, werden Daten mit Werten größer als '2004/01/01' und kleiner als '2005/01/01' eingefügt. Abbildung 23 zeigt die

Rückgabe der View *PartitionInfo* und dass diese Daten planmäßig in die Partition 3, somit auch in Filegroup FG3, gespeichert wurden. Die neu angelegte Filegroup ist zunächst leer, jedoch werden Daten, die einen Wert größer gleich '2005/01/01' besitzen, in die FG4 gespeichert. Dieser Prozess kann wiederholt werden, um weitere Partitionen zu fertigen.

Ergebnisse		Meldungen					
	Objekt_Name	partition_number	filegroup_name	rows	total_pages	Comparison	VALUE
1	myOrderTable	1	FG1	300	9	less than	2003-01-01 00:00:00.000
2	myOrderTable	2	FG2	300	9	less than	2004-01-01 00:00:00.000
3	myOrderTable	3	FG3	300	9	less than	2005-01-01 00:00:00.000
4	myOrderTable	4	FG4	0	0	less than	NULL

Abbildung 23: Rückgabe der View nach Erstellung der FG4 und dem Einfügen von Daten

Wenn beispielsweise die Anzahl der Partitionen sich auf eine Anzahl erhöht hat, bei der deren Verwaltung komplexer wird, gibt es die Möglichkeit, Partitionen zu löschen. Dies erfolgt, indem zwei Partitionen zu einer verschmolzen werden. Diese zusammengefügte Partition enthält dann die Inhalte beider Partitionen.

Im Anwendungsbeispiel werden die Partition 1 und die Partition 2 miteinander verschmolzen. Dies geschieht, indem der *Merge*-Methode der Parameter '2003/01/01', der die Grenze zwischen P1 und P2 darstellt, mitgegeben wird (Abbildung 24). Damit ändert sich die Partitionsfunktion, was zur Folge hat, dass diese Grenze aufgehoben wird und beide Partitionen miteinander fusionieren. Abbildung 25 zeigt, wie der ganze Inhalt von FG2 in FG1 übertragen wird. Filegroup FG2 wird hier nicht angezeigt, jedoch existiert diese immer noch, allerdings ohne Inhalt. Man könnte FG2 entfernen oder für andere Daten nutzen.

```
Alter Partition Function myPartitionFunction()
Merge Range ('20030201')
```

Abbildung 24: Merge-Funktion

Ergebnisse		Meldungen					
	Objekt_Name	partition_number	filegroup_name	rows	total_pages	Comparison	VALUE
1	myOrderTable	1	FG1	600	9	less than	2004-01-01 00:00:00.000
2	myOrderTable	2	FG3	300	9	less than	2005-01-01 00:00:00.000
3	myOrderTable	3	FG4	0	0	less than	NULL

Abbildung 25: Rückgabe nach Zusammenführung von Daten mittels Merge-Operators

In dem Fall, dass die Partition 1 mit den Daten aus 2002 und 2003 aus dieser Tabelle entfernt werden soll, wird der *Switch*-Operator verwendet. Unter den Einstellungen der Filegroups könnte man diese Daten, wenn nötig, auf *ReadOnly* setzen. Im folgenden Anwendungsbeispiel wird die Filegroup FG1 mit den Daten aus 2002 und 2003 aus dieser Tabelle entfernt. Der erste Schritt ist dabei, eine Tabelle *myOrderTableTMP* mit der gleichen Struktur der *myOrderTable* in der gleichen Filegroup zu erstellen, da Daten von Tabellen und Partitionen nur innerhalb der Filegroup geschoben werden können. Anschließend wird eine weitere Tabelle erstellt, die das eigentliche Ziel der Daten aus der Filegroup FG1 ist. Diese Tabelle *myOrderTableDestination* muss die gleiche Struktur wie die *myOrderTableTMP* und *myOrderTable* haben, jedoch spielt

es an dieser Stelle keine Rolle, wo sich die Tabelle befindet. Diese Tabelle könnte beispielsweise auch außerhalb der Datenbank liegen.

Nun kann mit dem *Switch*-Operator (Abbildung 26) die Partition 1 in die Tabelle *myOrderTableTMP* transportiert werden. Da an dieser Stelle nur Metadaten miteinander getauscht werden, erfolgt der Transfer augenblicklich.

```
Alter Table myOrderTable Switch Partition 1 to myOrderTableTMP
```

Abbildung 26: Partitionsfunktion Switch: Partition 1 in eine leere Tabelle

Um zu verifizieren, ob die Daten tatsächlich die Partition verlassen haben, hilft die View *PartitionInfo* (Abb. 27), dank der sichtbar ist, dass die Partition 1 und somit auch FG1 genau 0 Datensätze beinhaltet. Alle Daten aus dieser Partition werden nun von der Tabelle *myOrderTableTMP* gehalten.



	Objekt_Name	partition_number	filegroup_name	rows	total_pages	Compaction	VALUE
1	myOrderTable	1	FG1	0	0	less than	2004-01-01 00:00:00.000
2	myOrderTable	2	FG3	300	9	less than	2005-01-01 00:00:00.000
3	myOrderTable	3	FG4	0	0	less than	NULL

Abbildung 27: Rückgabe der View nach Nutzung des Switch-Operator

Im nächsten Schritt werden die Daten aus der temporären Tabelle (Abb. 28) mittels des *Insert*-Befehls in die Zieltabelle eingefügt. Der bereits angesprochene Vorteil dieser Methode liegt darin, dass hier keine langen *locks* entstehen. Das Verlagern der Daten mit dem *Switch*-Operator erfolgt lediglich auf der Metaebene, wodurch die Dauer des Vorgangs kurzgehalten wird. Daraus folgend ist die *myOrderTable* nur für eine kurze Zeit gesperrt. Die Daten aus der Tabelle *myOrderTableTMP* können nun unabhängig von der Quelltable *myOrderTable* in die Zieltabelle *myOrderTableDestination* verschoben oder gelöscht werden (Retama, 2011).

```
insert GrundlagenPartitionierung.dbo.myOrderTableDestination
(OrderID, OrderDate, Amount, ProduktID)
select * from myOrderTableTMP
```

Abbildung 28: Schieben der Daten von myOrderTableTMP zu myOrderTableDestination

2.3.6 Sliding-Window

Das Sliding-Window bezeichnet einen Prozess, in dem Partitionsoperationen in einer bestimmten Reihenfolge abgearbeitet werden. Der Grundgedanke besteht darin, in einer partitionierten Tabelle immer die gleiche Anzahl an Partitionen und Dateigruppen über einen bestimmten Zeitraum zu haben. Der Zyklus beginnt damit, dass die Datensätze der ältesten Partition gelöscht werden.

Bis zum SQL Server 2016 musste diese Partition, wie in Kapitel Partitionsfunktionen-*Switch* dargelegt, in eine andere Tabelle ausgelagert werden, die anschließend gelöscht werden konnte. Nach dieser Version ist es, wie in Abbildung 29 gezeigt, möglich, Datensätze einer kompletten Partition zu löschen (Sundar, 2018).

```
Truncate Table myOrderTable with (Partitions(@PartitionNumber))
```

Abbildung 29: Truncate-Funktion mit Angabe der Partition

Anschließend wird dem Partitionsschema die Dateigruppe mitgeben, die für die neueste Partition verwendet werden soll. Der nächste Schritt besteht darin, mit der *Split*-Funktion eine neue Partition zu erstellen. Dabei ist wichtig, dass Datensätze, die in diese Partition gespeichert werden, die gleiche Dateigruppe benutzen werden wie die älteste Partition, die im ersten Schritt gelöscht wurde. Da die älteste Partition keine Datensätze mehr enthält, kann diese nun mit dem *Merge*-Operator entfernt werden.

Im Anhang zeigt die Abbildung 65 die Schritte des *Sliding-Windows* gekapselt in einer *Stored-Procedure*, die ein Argument vom Datentyp *datetime* verlangt und diese als Grenzwert der neuen Partition nimmt (Sundar, 2018).

3 Ziele und Anforderungen

In diesem Kapitel werden Ziele und Anforderungen an die prototypische Implementierung behandelt. Ziele dieser Arbeit werden ermittelt und formuliert und anschließend werden die Anforderungen festgehalten. Zu den Anforderungen gehören Aspekte wie beispielsweise die Frage nach erwarteten Eigenschaften des fertigen Systems.

3.1 Ziele

In diesem Kapitel werden Ziele dieser Arbeit ausformuliert. Ziele geben an, was mit einem System erreicht werden soll. In Tabelle 9 werden sie in tabellarischer Form⁸ festgehalten. Die Tabelle enthält als Spalte die Nr. (Zielnummer) und die Beschreibung des jeweiligen Ziels. Die Zielnummer setzt sich aus dem Buchstaben Z zusammen, was im späteren Verlauf bei Verwendung dieser Nummer zu erkennen gibt, dass es sich um ein Ziel handelt. Zielnummern beginnen mit der Zahl 10 und steigen in Zehnerschritten auf.

Nr	Beschreibung
/Z10/	Steigerung der Systemperformanz
/Z20/	Die Basissoftware soll nicht verändert werden
/Z30/	Erfasste historische Daten müssen zur Laufzeit automatisiert archiviert werden können
/Z40/	Historische Daten müssen abrufbar und auswertbar sein
/Z50/	Archivierte Daten, die eine gewisse Lebenszeit erreicht haben, sollen gelöscht werden

Tabelle 9: Ziele

3.2 Anforderungen

In diesem Kapitel werden die Anforderungen an ein System festgehalten. Diese Anforderungen beschreiben, welche Dienste ein System mitbringen muss und was es leisten soll. Zudem beschreiben diese Anforderungen, wie ein System bei bestimmten Eingaben reagieren bzw. sich verhalten soll. Auch können Anforderungen an ein System Leistungen oder Dienste sein, die ein System nicht beinhalten soll. Die Anforderungen sind zum einen an die unterschiedlichen Benutzer gebunden und zu anderen an die Umgebung des Systems.

Im Folgenden werden die Anforderungen in Tabelle 10 festgehalten. Dabei wird zwischen den Spalten Kennzeichnung, Anforderung und Ziel unterschieden. Die Kennzeichnung identifiziert eine Anforderung, dient dem vereinfachten Referenzieren und erfolgt nach dem Schema /FXXXX/. Um spätere Änderungen an den Anforderungen vollziehen zu können, wird die Nummer (im Schema durch XXXX dargestellt) der Anforderungen in 10er-Schritten inkrementiert. Somit können später anfallende Anforderungen hinzugefügt werden. Der Präfix F ist eine Markierung für funktionale Anforderungen. Die Spalte Anforderung enthält eine kurze

⁸ Vgl. Balzert, H. (2009), Lehrbuch der Softwaretechnik, Basiskonzepte und Requirements Engineering

Beschreibung der Anforderung. Darüber hinaus enthält Tabelle 10 eine Spalte Ziel, welche die Anforderung einem oder mehreren vorher definierten Zielen zuordnet.

Kennzeichnung	Anforderung	Ziel
/F1010/	Beschreibung der Anforderung	Das referenzierte Ziel
/F1011/	Erweiterung der Anforderung /F1010/	Das referenzierte Ziel

Tabelle 10: Funktionale Anforderung Schablone

Die Anforderungen an das prototypisch zu implementierende Proof-of-Concept, die in Tabelle 11 dargestellt sind, umfassen von der Firma Köhl vorgegebene funktionale Anforderungen sowie auch Anforderungen an die Entwicklungsumgebung.

Kennzeichnung	Anforderung	Ziel
/F1010/	Daten müssen in Hot- und Cold-Data unterteilt werden	/Z10/
/F1020/	Hot- und Cold-Data müssen in Partitionen unterteilt werden	/Z10/
/F1021/	Partitionen müssen erweiterbar sein <ul style="list-style-type: none"> Erweiterung der Anforderung /F4020/ 	/Z10/
/F1022/	Partitionen müssen zusammengeführt werden können <ul style="list-style-type: none"> Erweiterung der Anforderung /F4020/ 	/Z10/
/F1023/	Partitionen müssen gelöscht werden können <ul style="list-style-type: none"> Erweiterung der Anforderung /F4020/ 	/Z10/
/F1030/	Daten müssen vorsortiert werden <ul style="list-style-type: none"> Erweiterung der Anforderung /F4020/ 	/Z10/
/F2010/	Funktionalitäten der Basis-Software dürfen nicht verändert werden	/Z20/
/F2011/	Datenbank muss als eine logische Einheit repräsentiert werden	/Z20/
/F2020/	Es muss für die objektorientierte Programmierung Java EE die IDE Eclipse verwendet werden	/Z20/
/F2030/	Die Logik muss auf der Serverseite entwickelt werden	/Z20/
/F2040/	Es muss die Microsoft SQL Server Datenbank verwendet werden	/Z20/
/F3010/	Daten müssen vom System erfasst werden können	/Z30/
/F3020/	Historische Daten müssen als solche gekennzeichnet werden	/Z30/
/F3030/	Historische Daten müssen als solche erfasst werden können	/Z30/
/F3040/	Daten müssen archiviert werden können	/Z30/
/F3050/	Historische Daten müssen zur Laufzeit archiviert werden können	/Z30/
/F3060/	Historische Daten müssen automatisiert ins Archiv verschoben werden können	/Z30/
/F4010/	Daten müssen für Klienten über einen Webservice bereitgestellt werden können	/Z40/
/F4011/	Der verwendete Webserver muss die JBoss-Enterprise-Application-Plattform sein <ul style="list-style-type: none"> Erweiterung der Anforderung /F4010/ 	/Z40/

/F4020/	Archivierte Daten müssen auswertbar sein	/Z40/
/F4030/	Archivierte Daten müssen in Auswertungen als solche sichtbar sein	/Z40/
/F4040/	Zugriffe auf die Datenbank müssen mittels Hibernate-CRUD-Operationen erfolgen	/Z40/
/F4041/	Create-Abfragen müssen getätigt werden können <ul style="list-style-type: none">• Erweiterung der Anforderung /F4040/	/Z40/
/F4042/	Read-Abfragen müssen getätigt werden können <ul style="list-style-type: none">• Erweiterung der Anforderung /F4040/	/Z40/
/F4043/	Update-Abfragen müssen getätigt werden können <ul style="list-style-type: none">• Erweiterung der Anforderung /F4040/	/Z40/
/F4044/	Delete-Abfragen müssen getätigt werden können <ul style="list-style-type: none">• Erweiterung der Anforderung /F4040/	/Z40/
/F5010/	Entitäten müssen ein Löschdatum enthalten	/Z50/
/F5020/	Das System muss das Löschdatum auswerten können	/Z50/
/F5030/	Das System muss die Daten aus dem Archiv löschen können	/Z50/

Tabelle 11: Anforderungen

4 Performanzsteigerung

In vielen Fällen kann die Leistung eines Systems gesteigert werden, indem das Design oder die verwendete Software angepasst wird. Demzufolge werden in den folgenden Kapiteln allgemeine Methoden und Techniken ausgearbeitet, die die Performanz eines Systems durch Designanpassung bzw. genutzte Software beeinflussen. Grundsätzlich ist zu konstatieren, dass es keine ultimative Lösung gibt, die alle Performanz-Probleme abdeckt. Methoden haben Vor- und Nachteile, die an den Anwendungsfall angepasst werden müssen. Dieses Kapitel gliedert sich in die Unterpunkte Designanpassung und Softwaretuning. Die Nutzung dieser Techniken zur Steigerung der Performanz eines Systems bedarf einer im Vorfeld durchgeführten ausgiebigen Analyse des Systems. Die Methoden sind nur dann nutzbar und voll einsetzbar, wenn das System in seiner Komplexität und samt seiner Schwachstellen bekannt ist. Ebenso muss bereits vor der Anwendung der Optimierungen ein klares Ziel für das System vorliegen (Rusanu Consulting, 2014).

4.1 Designanpassung

Im Kapitel Designanpassung werden Grundrichtlinien des Designs, mit denen bei richtiger Umsetzung das System in puncto Zugriff auf die Daten gesteigert werden kann, beschrieben. Das Kapitel unterteilt sich in Zugriffsmethoden, Indexierung, Datenbankdesign und Wahl der Datenbank. Im ersten Kapitel geht es um Strukturen, also wie man Daten in eine Form bzw. Vorlage abspeichern kann. Eine Form, wie man Daten aufbereiten bzw. ablegen kann, um einen schnelleren Zugriff zu bekommen, wäre die Nutzung sogenannter Sichten (Views) und materialisierte Sichten (Materialized-Views). Im Kapitel Index werden Sortierstrukturen in relationalen Datenbanken untersucht. Eine geeignete Wahl der Sortierung sorgt dafür, dass Resultate von Anfragen effizienter zurückgeliefert werden. Im Anschluss folgt das Kapitel Datenbank-Design, in dem es darum geht, den Aufbau und die Struktur einer Datenbank, den Wünschen der Software und der Anwendungsfälle so anzupassen, dass die Anfragen performanter ausgeführt werden. An dieser Stelle werden auch die Themen Denormalisierung und Normalisierung einander gegenübergestellt und diskutiert. Die Wahl der Datenbank ist das letzte Kapitel der Designanpassung, hier werden Vor- und Nachteile aufgelistet, wann welches Datenbankmodell für welchen Fall geeignet ist.

4.1.1 Index

Ein Index beschreibt die Struktur, wie Daten auf einem Datenträger abgelegt werden. Ein Index kann einer Tabelle oder einer View zugeordnet werden. Richtig eingesetzt können Indizes dafür sorgen, dass das Abrufen von Datensätzen aus Tabellen bzw. Views beschleunigt wird. Aus einer oder mehreren Spalten wird ein Schlüssel erstellt, der dem Index zugeteilt wird. Anhand dieses Schlüssels kann der SQL Server Datensätze, die dem Schlüssel zugeordnet sind, effizienter finden. Im Grunde werden drei Kategorien unterschieden: Heap, gruppiert und nicht gruppiert.

Eine Tabelle, die keinen Primary-Key und keinen Index enthält, hat keine Struktur und wird Heap genannt. Wenn Daten in die Tabelle eingefügt werden, werden diese abhängig vom freien Speicherplatz und nicht nach einer Reihenfolge eingefügt. Wenn eine Suche ausgeführt

werden soll, wird die komplette Tabelle durchlaufen, auch wenn bereits ein Treffer gefunden wurde. Es könnte nämlich sein, dass ein weiterer Datensatz auf die geforderte Abfrage passt. Diese Art der Suche nennt man *Scan* und sie wird als ineffizient eingestuft.

Eine Tabelle mit einem Index sorgt dafür, dass eine Sortierung der Daten anhand von einer oder mehrerer Schlüsselspalte(n), die den Index bilden, stattfindet. Eine Sortierung könnte beispielsweise nach einer Tabellenspalte ID, Name oder Datum erfolgen. Abfragen werden, wie beim Heap, mittels *Scan* durchgeführt, mit dem Unterschied, dass sie nicht zwangsläufig alle Daten untersuchen müssen. Weil die Daten sortiert vorliegen, bricht die Suche ab, sobald die geforderten Informationen gefunden wurden. Durch die Sortierung wird verhindert, dass weitere Datensätze an einer anderen Stelle ebenfalls mit der Abfrage übereinstimmen.

Mit einem gruppierten Index (Clustered-Index) werden die Daten abhängig von den Schlüsselwerten sortiert. Wenn ein Primär-Schlüssel erzeugt wird, wird in SQL Server standardmäßig ein gruppierter Index mit der entsprechenden Spalte als Schlüssel angelegt. Zusätzlich erzeugt SQL Seiten, die Informationen zum Index beinhalten. Mithilfe dieser Seiten können Anfragen nicht relevante Tabellen bzw. Datensätze ausschließen und zum Ergebnis navigieren. Die Struktur der erzeugten Seiten ist in Form eines Baums angelegt, wobei die Daten, die physisch auf dem Datenmedium liegen, die Blätter sind. Diese Art der Suche über den Baum zu den Blättern, wird als *Clustered-Index-Seek* bezeichnet. Ein Telefonbuch ist ein praktisches Beispiel für diese Art der Suche. Ein Telefonbuch ist alphabetisch nach Namen sortiert. Somit hat der Index das Schlüsselattribut Name. Wenn die Telefonnummer von Herrn Tutzauer gesucht wird, würden alle Kapitel von A bis zum Anfangsbuchstaben T übersprungen. Es würde dann so lange geblättert, bis der Zweitbuchstabe *u* kommt usw., bis Herr Tutzauer und seine Telefonnummer gefunden sind. Da die Sortierung auf den physischen Daten beruht, ist nur eine Sortierung möglich und schlussfolgernd kann auch nur ein Clustered-Index für eine Tabelle existieren.

Bei der Nutzung von nicht gruppierten Indizes (Non-Clustered Indizes) werden die ursprünglichen Daten nicht sortiert. Die Struktur der Datensätze in den Tabellen ist ein Heap, also ungeordnet. Jedoch werden Index-Pages (Seiten) in einer Baumstruktur erzeugt, die einem Inhaltsverzeichnis ähneln. Mit dem Inhaltsverzeichnis ist es möglich, durch den Baum effizient Datensätze zu finden. Im Gegensatz zum Clustered-Index enthalten alle Seiten wie auch die Blätter nicht die tatsächlich physischen Daten, die auf einem Datenmedium liegen, sondern nur einen Zeiger, der auf einen Datensatz im Heap verweist. Non-Clustered Indizes sind von den tatsächlichen Tabellen separiert und müssen im Gegensatz zum Clustered-Index nicht alle Daten beinhalten. Abgesehen vom *Explizit*-Operator wird auch durch die Verwendung des *UNIQUE*-Befehls automatisch ein nicht gruppierter Index angelegt.

Wenn eine Anfrage an eine Tabelle getätigt wird, die über nicht gruppierte Indizes verfügt, wird der Query-Optimizer diese nutzen, um das Ergebnis der Anfrage zurückzugeben. Wenn diese Art der Datensuche jedoch für den Query-Optimizer nicht performant genug ist, wird sie verworfen werden und stattdessen ein *Full Scan* durchgeführt.

In der Regel wird eine Kombination aus Clustered und Non-Clustered Indizes angestrebt, wobei der Clustered-Index die Sortierung der physischen Daten vorgibt und der Non-Clustered Index als Hilfe genutzt wird, falls Datensätze aus Spalten abgefragt werden, die den Clustered-Index nicht verwenden (**Microsoft SQL Dokumentation, 2019**).

4.1.2 Materialisierte- und indizierte Sichten

In relationalen Datenbanken werden Tabellen genutzt, um Daten darin zu speichern. Wie oben erläutert, können Datenbanken viele Tabellen mit einer Menge von Datensätzen enthalten, die durch Assoziationen miteinander verbunden sind. Mit *Select*-Befehlen in Kombination mit Operatoren wie Aggregatsfunktionen, Mengen-Operatoren und Joins werden diese Tabellen miteinander verbunden, durchsucht und man erhält das Ergebnis einer Anfrage. Diese Anfragen

(Queries) können sehr umfangreich und dadurch auch performanzlastig werden. Datenbanksysteme bieten unterschiedliche Arten von Views an, um dem entgegen zu wirken. Die Oracle-Datenbank verfügt über die klassische View wie auch über die materialisierte View (MAV). Seit der SQL Server 2000 Version wurde die klassische View um Indizes zu Indexed-Views (Indizierte Sichten) erweitert.

Eine View ist eine Anfrage, die in der Datenbank gespeichert wird. Somit kann man auf die View zurückgreifen, um Anfragen an die Datenbank zu verschicken. Diese View vereinfacht Queries, da man sie nicht erneut neu schreiben muss, jedoch haben sie die gleiche Laufzeit wie manuell gesendete Anfragen. In Oracle wie auch MSSQL Server werden die Daten der View nicht in der Datenbank gespeichert, sondern nur die angegebene Query (Brumm, 2018). Durch das Einsetzen von Materialized-Views in der Oracle-Datenbank kann im Gegensatz zu einer View, die nur den Befehl speichert, das ganze Ergebnis in Form einer Tabelle deponiert werden. Abfragen mit komplexen Berechnungen über mehrere Tabellen und Aggregatsfunktionen können in so einer Materialized-View abgespeichert werden. Das Resultat dieser Abfrage wird ähnlich wie eine Tabelle in der Datenbank aufbewahrt. Ein Aufruf dieser Materialized-View wird das gleiche Ergebnis zurückliefern wie die entsprechende Query, jedoch mit einem bemerkbaren Performanzgewinn, da die Berechnungen bereits vorliegen und nicht mehr durchgeführt werden müssen.

Da diese spezielle OracleDB-View in die Datenbank gespeichert wird, entsteht ein Performanzgewinn, der jedoch zulasten des Speicherplatzes geht. Zusätzlich muss man in Kauf nehmen, dass die MAV eine Art Schnappschuss der aktuellen Daten darstellt. Im Fall einer Aktualisierung oder Änderung von Tabellen, aus der die MAV ihre Daten bezogen hat, würde sich die MAV nicht aktualisieren.

Um die Daten weiterhin konsistent zu halten, müsste man die MAV aktualisieren und mit aktuellen Daten neu laden. Abhängig davon, wie oft Tabellen, die Datensätze einer MAV enthalten, sich ändern, würde man an dieser Stelle einen Trigger implementieren. Ein Trigger ist eine Funktion, die dann vom DBMS gefeuert wird, wenn eine bestimmte Funktion ausgelöst wird. Wenn eine dieser Basistabellen mit einem *INSERT*, *UPDATE* oder *DELETE* geändert werden sollte, würde dies einen Trigger werfen, der automatisch die Materialized-View dementsprechend anpasst und aktualisiert. In Abbildung 30 ist ein Code-Beispiel, wie eine Materialized-View in Oracle erstellt werden würde und wie auf diese zugegriffen werden könnte.

```
CREATE MATERIALIZED VIEW myMaterializedView
[REFRESH [FAST|COMPLETE|FORCE] [ON DEMAND|ON COMMIT]]
[BUILD IMMEDIATE|BUILD DEFERRED]
AS
SELECT a.column1, a.column2, a.column3,
       b.column1, b.column2;
FROM TableName1 a
INNER JOIN tableName2 b ON a.table1 = b.table1

SELECT * from myMaterializedView
```

Abbildung 30: Materialized-View erstellen in Oracle (Oracle Help Center, n.d.)

Die Indexed-View von SQL Server ist eine View, die einen Unique Clustered-Index hat. Sobald eine View solch einen Index bekommt, wird diese View materialisiert und in der Datenbank persistent gespeichert. Ergebnisse von Abfragen, die auf Spalten mit komplexen Berechnungen und Aggregatsfunktionen abzielen, können auf diese Weise vorberechnet werden und wie bei der Materialized-View von der Oracle Datenbank performant und effizient abgerufen werden. Im Fall, dass die Indexed-View alle Daten oder einen Teil der geforderten Daten enthält und

eine schnellere Rückgabezeit hat als der eingegebene SQL-Befehl, wird der SQL Server Query auf die Indexed-View zugreifen, und zwar ohne eine explizite Angabe des Benutzers. Abbildung 67 im Anhang zeigt ein Beispiel, wie eine erstellte Indexed-View verwendet werden kann, ohne dass sie in der *From*-Klausel explizit aufgerufen wird (**Microsoft SQL Dokumentation, 2018**).

Die Performanz der Indexed-View kann durch den Einsatz von Non-Clustered Indizes weiter verbessert werden, denn durch die weitere Angabe von Non-Clustered Indizes hat der SQL-Query-Optimizer mehr Möglichkeiten, schneller an die Daten zu gelangen, sie zurückzuliefern, womit er gegebenenfalls einen *Full Scan* vermeidet. Im Gegensatz zu der Materialized-View müssen die Indexed-Views nicht manuell aktualisiert werden. Der Query-Plan ändert die Basistabellen und synchronisiert sie mit der IndexedView.

Jedoch führt dies auch dazu, dass die ständige Aktualisierung einen starken Overhead mit sich bringt. Daher sollen die Indexed-Views nur für Tabellen verwendet werden, die eher statisch sind und nicht oft geändert werden. Zudem ist drauf zu achten, Indizes nur dann zu verwenden, wenn diese auch genutzt werden. Ansonsten hätte man zum einen den extra generierten Overhead, der durch die ständige Wartung entsteht, und zum anderen verschwendeten Speicherplatz und damit auch keine Performanzverbesserung.

Um diese Arten der Views effizient zu verwenden, ist es daher wichtig zu wissen, welche Queries und wie oft diese verwendet werden. Grundsätzlich geht jede Verbesserung auch mit einem negativen Aspekt einher und man muss stets den besten Weg abwägen. So ist Materialized-View zwar nützlich, aber auch umständlich, denn sie muss stets konsistent gehalten werden. Indexed-View wiederum kann unter Umständen durch die Indizes einen großen Verwaltungsaufwand mit sich bringen (Yaseen, 2016).

Zusätzlich sollte man wissen, ob diese Queries komplex genug sind, sodass man den Trade-off von Performanz zu Speicherkapazität und dem damit einhergehenden Verwaltungsaufwand der Indizes abwägen kann.

4.1.3 Datenbankdesign

Ein guter Datenbankentwurf zeichnet sich dadurch aus, dass er wenige Redundanzen aufweist. Die selben Daten sollen nicht an mehreren Plätzen vorhanden sein. Zum einen würde dies zu Inkonsistenz führen und zum anderen wäre es eine Verschwendung von Speicherplatz. Dieses Kapitel befasst sich mit dem Entwurf einer Datenbank und den Fallstricken, die mit einem gut gewählten Entwurf vermieden werden können.

Um den Entwurf einer Datenbank zu erstellen, muss man sich Gedanken machen über die Daten, die gespeichert werden sollen. Was haben diese für einen Datentyp und wie sind sie miteinander verbunden? Zudem muss die Frage beantwortet werden, welche Funktionen von einer Datenbank erwartet werden und welche Art von Anfragen diese Datenbank verarbeiten soll. Im Grunde entsteht somit ein Datenbank-Entwurfs-Zyklus, der in Abbildung 31 dargestellt ist.



Abbildung 31: Datenbank-Entwurfs-Zyklus

Wenn man mit Tabellen arbeitet, die verschiedene Spalten aufweisen wie etwa Name, Titel oder Adresse, kann sich die Sortierung der Daten als problematisch erweisen. Beispielsweise ist es regional abhängig, in welcher Reihenfolge man Namen abspeichert, es ist entweder Name, Vorname oder Vorname Name. Sollte der Befehl verlangen, die Spalte Name alphabetisch zu ordnen, wäre eine solche Sortierung ohne komplexe Zeichenketten-Manipulations-Operatoren an dieser Stelle nicht möglich. Daher wird die Spalte Name in die Spalten Vorname und Nachname unterteilt. Dies würde für die Spalte Adresse ebenfalls gelten. Diese würde man in Straße, Hausnummer, PLZ und Ort unterteilen.

Wenn nun weitere Daten bestimmt werden, z. B. die Bestellung eines Kunden mit den Spalten Bestellnummer, Produkte, Produktpreis und Gesamtpreis, muss definiert werden, wie diese Spalten mit den vorherigen Daten in Verbindung stehen. Eine Erweiterung der ersten Tabelle würde an dieser Stelle keinen Sinn ergeben, da ein Name mehrere Bestellungen tätigen kann, wobei eine Tabelle mehrere Datensätze aufweisen würde mit Titel, Name, Adresse. Um die Redundanzen zu eliminieren, würden diese Spalten in eine weitere Tabelle ausgelagert werden, die nur die Bestelldaten enthält, dies mit einer Referenz zur Kunden-Tabelle.

Spalten besitzen Datentypen. Zeichenketten (Strings) und Zahlen (Integer) kann man auf eine gewisse Anzahl Zeichen begrenzen. Jedoch sollte man dies nicht tun und die Begrenzung, wenn die Möglichkeit besteht, auf das Maximum einstellen, denn mit der Zeit können vorher fix gewählte Begrenzungen evtl. nicht mehr ausreichend sein. Die Geschäftslogik sollte die Begrenzungen übernehmen, die Aktionen des Benutzers lenken und den eingegebenen Input kontrollieren.

Ein weiterer Punkt ist die Wahl der Namensgebung der Spalten. In der Regel sollte man Leerzeichen, obwohl sie eine Spalte z. B. *Order Details* leserlicher machen, vermeiden, denn es könnte sein, dass man in Zukunft mit einem anderen System zusammenarbeitet, das keine Leerzeichen zwischen Zeichenketten erlaubt.

Im Gegensatz zum Datentyp *Date*, der von SQL standardisiert wurde, existiert kein Datentyp für Geldwert. In der Regel wird ein Geldbetrag in einem Dezimalwert (*float*) mit zwei Kommastellen angegeben, was zu Rundungsfehlern führen kann. Hier sollte ein Dezimalwert mit mehr als fünf Nachkommastellen genommen werden. Um der Währungsproblematik entgegenzuwirken, sollte zusätzlich eine weitere Tabelle mit den Währungen erstellt werden, die mittels Fremdschlüsseln verbunden werden.

Spalten, die oft verwendet oder aktualisiert werden, wie Telefonnummer oder Kreditkartennummer, sollten nicht in jede Tabelle gespeichert, sondern in eine separate Tabelle ausgelagert werden und mit einem Fremdschlüssel verbunden sein. Dies würde gewährleisten, dass die Daten redundanzfrei bleiben und zudem müsste eine Datenänderung nur an einer Stelle getätigt werden. Diese Art der Aufteilung der Daten in ihre Einzelteile und deren Verbund (mit Fremdschlüsseln) ist auch bekannt als Normalisierung (Dartmouth.edu, 2004). In Tabelle 12 sind fünf Schritte angegeben, die durchgeführt werden müssen, um Daten in normierter Form zu bekommen, wobei die dritte Normalform in der Regel ausreichend ist, um die Datenbank redundanzfrei zu gestalten (Begerow, Datenbanken-Verstehe.de, 2019).

Nr.	Beschreibung
1NF	Die erste Normalform ist dann gegeben, wenn alle Informationen in einer Tabelle atomar vorliegen (Begerow, Datenbanken-Verstehen.de, 2019).
2NF	Ein Relationstyp (Tabelle) befindet sich genau dann in der zweiten Normalform, wenn er sich in der ersten Normalform befindet und jedes Nichtschlüsselattribut von jedem Schlüsselkandidaten voll funktional abhängig ist (Begerow, Datenbanken-Verstehen.de, 2019).

3NF	Ein Relationstyp befindet sich genau dann in der dritten Normalform, wenn er sich in der zweiten Normalform befindet und kein Nichtschlüsselattribut transitiv von einem Kandidatenschlüssel abhängt (Begerow, Datenbanken-Verstehen.de, 2019).
4NF	Ein Relationstyp befindet sich genau dann in der vierten Normalform, wenn er sich in der Boyce-Codd-Normalform befindet und für jede mehrwertige Abhängigkeit einer Attributmenge Y von einer Attributmenge X gilt: – Die mehrwertige Abhängigkeit ist trivial oder – X ist ein Schlüsselkandidat der Relation (Begerow, Datenbanken-Verstehen.de, 2019).
5NF	Ein Relationstyp befindet sich genau dann in der fünften Normalform (5NF), wenn er sich in der vierten Normalform (4NF) befindet und für jede Abhängigkeit (R_1, R_2, \dots, R_n) gilt: – Die Abhängigkeit ist trivial oder – jedes R_i aus (R_1, R_2, \dots, R_n) ist Schlüsselkandidat der Relation (Begerow, Datenbanken-Verstehen.de, 2019).

Tabelle 12: Die fünf Normalformen

Nachdem die Daten wie in Abbildung 31 bestimmt, erstellt und nach Tabelle 12 normalisiert wurden, werden im Entwurfsprozess Primär sowie auch Fremdschlüssel gesetzt. Schlüssel in Tabellen sorgen dafür, dass jeder Datensatz eindeutig identifiziert werden kann und sind somit auch ein Werkzeug, das verwendet werden kann, um die Integrität der Daten zu gewährleisten. Beispielsweise ist es möglich, mithilfe der Schlüssel dafür zu sorgen, dass in zwei Spalten verschiedener Tabellen der gleiche Inhalt steht. Sie werden ebenfalls verwendet, um Abhängigkeiten zwischen Tabellen herzustellen. Der wichtigste Schlüssel ist der Primärschlüssel (Primary-Key). Durch diesen wird in der gesamten Datenstruktur eine Tabelle eindeutig identifiziert und Datensätze werden eindeutig repräsentiert. Damit wird die Konsistenz der Daten gewahrt.

Die Wahl eines guten Primärschlüssels ist maßgeblich und wichtig. Bei der Wahl, welche Spalte bzw. Spalten einem Primary-Key zugeordnet werden, ist zu beachten, dass die Wahl kein mehrteiliges Feld ist, sondern eindeutige Werte besitzt. Zudem muss in jedem Feld ein Wert enthalten sein und es dürfen keine Null-Werte in der Spalte vorkommen. Der Primärschlüssel muss den vollständigen Datensatz eindeutig identifizieren können. Auch kann der primäre Schlüssel aus einem Verbund aus mehreren Spalten bestehen.

Der Fremdschlüssel ist eine Spalte in einer Tabelle, die auf den primären Schlüssel einer anderen (Tabelle) oder auch der eigenen Tabelle verweist. Mit dem Fremd- und dem Primärschlüssel wird die Beziehung zwischen zwei Tabellen hergestellt. Der Fremdschlüssel kann aus einer Spalte oder aus mehreren Spalten zusammengesetzt sein. Die Werte der Fremd- und Primärschlüsseln müssen, um die Konsistenz zu bewahren identisch sein (Dartmouth.edu, 2004).

Es gibt drei Arten der Beziehungen zwischen Tabellen: One-To-One, wo einem Datensatz genau ein weiterer Datensatz einer anderen Tabelle zugewiesen wird, One-To-Many (und vice versa), wo einem Datensatz eine Collection (Liste) an Datensätzen einer anderen Tabelle zugesprochen wird und Many-To-Many (Beziehungen), wo einem Datensatz eine Collection (Liste) an Datensätzen zugewiesen wird und dieser ein Datensatz in einer anderen Collection (Liste) ist. Diese Art der Beziehungen wird mittels einer Hilfstabelle realisiert. Die Hilfstabelle enthält nur Fremdschlüssel und evtl. auch andere Spalten, die Daten enthalten. In dieser Hilfstabelle setzt sich der primäre Schlüssel aus den Spalten aller Fremdschlüssel zusammen. Somit wird der komplette Datensatz eindeutig.

Wenn in normalisierten Datenbankmodellen Basis-Datenbestände voneinander getrennt in verschiedenen Tabellen gespeichert werden, um die Integrität und Konsistenz der Daten zu gewährleisten, werden *JOIN*-Operatoren genutzt, um diese Daten wieder zusammenzufügen. Durch die Normalisierung der Daten wird die Komplexität in Datenbanken künstlich erhöht und die Lesegeschwindigkeit durch *JOIN*-Operatoren verringert.

Um die Komplexität und vor allem die Performanz in puncto Lesegeschwindigkeit zu steigern, ist es möglich, die Daten gezielt, abhängig vom Anwendungsfall, zu denormalisieren. Um jedoch effizienten Gebrauch von der Denormalisierung zu machen, muss bekannt sein, um

welche Art von Daten es sich handelt und wie diese verwendet werden. Wenn in einem System die Attribute Titel, Vorname und Nachname nur in Kombination auftreten und eine separate Verwendung nicht benötigt wird, könnte man von der ersten Normalform abweichen und diese drei Spalten zu einer zusammenführen (Dartmouth.edu, 2004).

Wenn man von der zweiten und dritten Normalform abweicht und in diesem Sinne denormalisiert, zielt man drauf ab, die *JOIN*-Operatoren, um die einzelnen Tabellen zu verbinden, zu minimieren und damit die Lesegeschwindigkeit zu erhöhen. In der Regel zielt diese Art der Denormalisierung auf Tabellen mit One-To-Many-Beziehungen ab. Wenn es gewünscht ist rauszufinden, in welcher Abteilung ein Mitarbeiter arbeitet, würde man im Normalfall die Tabellen *Mitarbeiter* und *Abteilung* mit einem *JOIN*-Operator verbinden, um die Zuordnung Mitarbeiter zu Abteilung herzustellen. Denormalisiert könnte man die Abteilung des Mitarbeiters mit in die Mitarbeitertabelle als Spalte integrieren. Somit würde die Tabelle *Mitarbeiter* ausreichen, um die gewünschte Information zu bekommen (Wikipedia.de, 2019).

Das Schneeflockenschema, welches eine Weiterführung des Sternschemas ist, nutzt die Denormalisierung, um Lesezugriffe performanter zu machen. Dieses Datenmodell besteht aus einer Faktentabelle und weiteren Dimensionstabellen. Die Faktentabelle enthält Kennzahlen, Messwerte, Fremdschlüssel zu den Dimensionstabellen und wird in Darstellungen immer mittig platziert. Der primäre Schlüssel der Faktentabelle bildet sich durch die Kombination aus allen Fremdschlüsseln. Die Dimensionstabellen beinhalten Daten, die die Werte der Faktentabelle beschreiben. Diese Dimensionen werden grafisch um die Faktentabelle herum verstreut modelliert. Zusätzlich werden Dimensionstabellen weiter in untergeordnete Tabellen strukturiert, somit entsteht bildlich gesehen die Form einer Schneeflocke.

Die Dimensionstabellen liegen denormalisiert vor und werden mit jeder weiteren untergeordneten Stufe kleiner und kleiner. Abhängig davon, aus welcher Tiefe der Schneeflocke die Daten benötigt werden, braucht man unter Umständen Joins, um die Daten zusammenzuführen (Wikipedia.de, 2018). Diese Art der Modellierung findet Anwendung in Data-Warehouses, wo mehr Wert auf die Performanz des Lesezugriffs gelegt wird. Der Nachteil der Denormalisierung liegt im Bedarf nach zusätzlichem Speicherplatz durch die redundanten Daten und der Schwierigkeit, die Daten konsistent zu halten. Somit gilt es abzuwägen, ob der Speicherkostenfaktor und der zusätzliche Verwaltungsaufwand (um die Daten konsistent zu halten) die Verminderung der Komplexität und Erhöhung der Lesegeschwindigkeit rechtfertigen (Bernhard Lahres, 2006).

4.1.4 Wahl der Datenbank

Die Wahl, welches Datenbankmodell genutzt werden soll, ist eine fundamentale Entscheidung. Abhängig von der Datenstruktur und dem Gebrauch der Daten muss überlegt werden, ob eine relationale Datenbank, eine dokumentenbasierte oder eine objektorientierte Datenbank (Objektdatenbank) geeignet ist. In Kapitel 2.1 wurden die Grundlagen der drei Datenbankmodelle ausgearbeitet. In diesem Kapitel wird auf die Vor- und Nachteile der Datenbankmodelle eingegangen bzw. auf die Frage, welches Modell sich in bestimmten Anwendungsfällen besser eignet als die anderen. Das Hauptaugenmerk bleibt die Performanz mit Zugriff auf Daten.

In Abbildung 32 und 33 werden Benchmarktests von Objektdatenbanken und relationalen Datenbanken gezeigt. Dabei gilt: Je höher der Wert auf der Y-Achse ist, desto besser ist das Ergebnis. Die Werte wurden nach Formel 1 normiert, um ein vereinfachtes Resultat zwischen 0 und 100 zu erzielen.

$$100 * \{\text{absolutes Resultat}\} / \{\text{best erzielt es absolutes Resultat}\}$$

Formel 1: Berechnung der Normierung aus (JPA Performance Benchmark (JPAB), 2012)

Dabei trägt *absolute Resultat* den Wert der eigentlichen zu testenden Datenbank und *best erzieltes absolute Resultat* das beste Ergebnis aller getesteten Datenbanken. Das Ergebnis wird durch Tests in sechs Bereichen ermittelt. Jeder Test überprüft die CRUD-Operationen und deren Speichereffizienz in Bezug auf die Datenbankgröße. Zudem wird jede Operation in Bezug auf Datenabrufe und Transaktionsgröße gegen viele und wenige Daten geprüft. Die Kombination aller Testmöglichkeiten sind 58 Leistungstests und sechs Tests zur Prüfung der Effizienz der Speicherung. Den gesamten Vergleich findet man unter JPA Performance Benchmark (JPA Performance Benchmark (JPAB), 2012).

Relationale Datenbank

Relationale Datenbanken sind, wenn die Daten normiert wurden (vgl. Tab. 9), redundanzfrei und demzufolge auch konsistent. Ein großer Vorteil bezüglich der Effizienz bei der Aktualisierung der Daten, der sich ebenfalls in der Performanz ausdrückt, ist die sogenannte Redundanzfreiheit. Dadurch ist sichergestellt, dass ein Datenelement nur an einer Stelle in der Datenbank vorhanden ist und somit auch nur an dieser Stelle geändert werden muss (Amdtown, 2019).

Durch das einfache Datenmodell und die garantierte Konsistenz der Daten ist der Umgang mit dem System einfacher und leichter zu verstehen. Zudem wird nicht durch die Struktur der Daten des Systems navigiert, um die Daten zu bekommen, und man muss sich keine Gedanken darüber machen, *wie* die Daten beschafft werden, da das DBMS das übernimmt. Der Benutzer drückt durch eine Anfrage über die standardisierte SQL-Sprache an das DBMS aus, *was* verlangt wird, und bekommt die geforderten Daten zurück (Mutschke, 1995). Mit der Sprache SQL werden die Datenmanipulationsfunktionen der RDBMS umgesetzt. Da das relationale Datenmodell das am weitesten verbreitete und meist genutzte Datenbankmodell ist, wurde es für sämtliche Aspekte, wie die Leistungssteigerung der Datenbank, genutzte Technologie und Speicherverbrauch, optimiert (Amdtown, 2019). Beispielsweise bieten Datenbanksysteme wie der SQL Server sogenannte Query-Optimizer an, die in gegebenen Situationen inperformante Queries verbessern und das geforderte Ergebnis zurückliefern. Zudem erlaubt es ein sogenannter Execution-Plan, im SQL Server nachzuvollziehen, wie die Daten vom DBMS aus der Datenbank entnommen bzw. zusammengestellt wurden und ermöglicht weitere Optimierung.

Die Performanz kann bei einem System, das zwischen einer objektorientierten Unternehmenssoftware und einer relationalen Datenbank kommuniziert, mithilfe eines ORM-Frameworks wie Hibernate (vgl. Kap 2.2.2) gesteigert werden.

In Abbildung 32 wird das Framework Hibernate in Kombination mit diversen Datenbanken in Bezug auf Performanz (grün dargestellt) und Speicher in einem Benchmarktest verglichen.

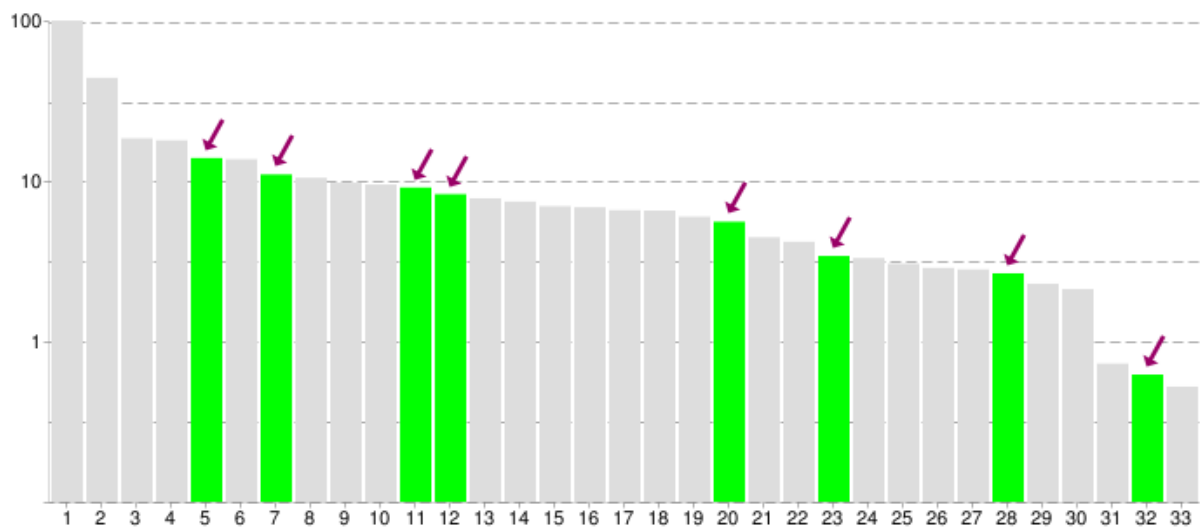


Abbildung 32: Leistung Hibernate (grün) im Vergleich zu anderen JPA/DBMS-Kombinationen (grau) (<http://www.jpab.org/Hibernate.html>)

Objektorientierte Datenbanken

Während sich die relationale Datenbank bei normalen Anwendungen gut bewährt, stößt sie mit der zunehmenden objektorientierten Programmierung an ihre Grenzen. Dementsprechend wurde das Angebot der Objektdatenbanken immer größer. Objekte, die in eine Datenbank persistent gespeichert werden müssen, werden komplizierter. So kommen zudem noch Vererbungshierarchien und vermehrt Relationen wie die Many-To-Many-Abhängigkeit (Gebhardt, 2018) hinzu. OBD lösen nativ die Probleme, mit denen relationale Datenbanken Schwierigkeiten haben.

Ein charakteristisches Merkmal der OBD ist es, dass der Entwickler der Objektdatenbank die Freiheit und Flexibilität hat, Objektstrukturen sowie auch Operatoren, die auf diesen angewendet werden, selbst zu bestimmen und zu definieren.

In Objektdatenbanken werden die Daten in Form von Objekten mit sämtlichen Attributen gespeichert. Bei einer Abfrage an ein Objekt kann man sich alle Attribute ansehen und erhält all ihre Informationen. Im Gegensatz zu relationalen Datenbanken, wo die Daten nach Eigenschaften gruppiert in Tabellen vorliegen, muss man verschiedene Tabellen mit *JOIN*-Operatoren verknüpfen, um alle Werte eines Objekts zu erhalten. In Audit-Fällen, wo alle Eigenschaften eines Objekts in Betracht gezogen werden, wäre die Nutzung einer OBD von Vorteil, da die Daten nicht durch *JOIN*-Operatoren zusammengefügt werden müssen. Doch im Fall eines ständigen Betriebs, in dem nicht alle Attribute eines Objekts verlangt werden, kann es von Vorteil sein, durch Relationen auf einzelne gebrauchte Attribute zuzugreifen.

Wenn nun ein objektorientiertes System mit einer relationalen Datenbank zusammenarbeiten soll, muss man sich mit dem relationalen Bruch auseinandersetzen. Durch die Speicherung der Objekte in ihrer Ursprungsform in Objektdatenbanken entfällt diese Überlegung, womit die Komplexität der Programmierung reduziert wird und auch die Datenbankabfrage schneller erfolgt. Es muss keine Lösung selbst entwickelt oder ein Hilfsframework zur Hilfe genommen werden und es kann sich vollkommen auf die Geschäftslogik konzentriert werden. So kann die Firmensoftware auf einer Ebene mit der Objektdatenbank kommunizieren.

In RDBMS werden komplexe Objekte durch Tabellen und Hilfstabellen, die mit Relationen verbunden sind, abgebildet. Dies bedeutet immer einen Mehraufwand. Mit Objektdatenbanken können diese komplexen Objekte auf natürliche Art in der Datenbank gespeichert werden. Darüber hinaus können auch Datenstrukturen wie die Vererbungshierarchien einfach

abgebildet werden. Die auf der OBD gespeicherten Objekte werden in dem verwendeten Format (Java oder C++) abgespeichert. Da die Objekte in der ausgewählten Programmiersprache gespeichert werden, ist keine Übersetzung der Daten erforderlich, was wiederum eine Leistungssteigerung gegenüber einer RDM vom Faktor 10 bis 100 – abhängig von der Komplexität der Objekte – ermöglicht (Barry, Service-Architecture, 2010 - 2019).

Zudem sind im Datenbankmanagement-System semantische Zusammenhänge zwischen Objekten durch die komplette Speicherung der Struktur bekannt. Das heißt, dass das DBMS die Zusammenhänge von Objekten bei Abfragen kennt und den Benutzer mit Hinweisen unterstützen kann.

Objektdatenbanken haben jedoch noch immer Nachteile gegenüber relationalen Datenbanken, etwa bei der Verarbeitung großer Datenmengen. Dies wird beispielsweise durch Zugriffspfade zu Objekten über mehrere Pfadarten (bspw. Vererbung und Assoziation) verursacht. Dies führt bei Schreiboperationen in der Sperrverwaltung zu einer exponentiellen Komplexität und somit zu schlechterer Leistung (Wikipedia, 2018).

In Abbildung 33 ist zu sehen, wie die Objektdatenbank ObjektDB (grün) hinsichtlich Performanz im Vergleich zu anderen Datenbanken (grau), die die Schnittstelle JPA implementieren, verhält.

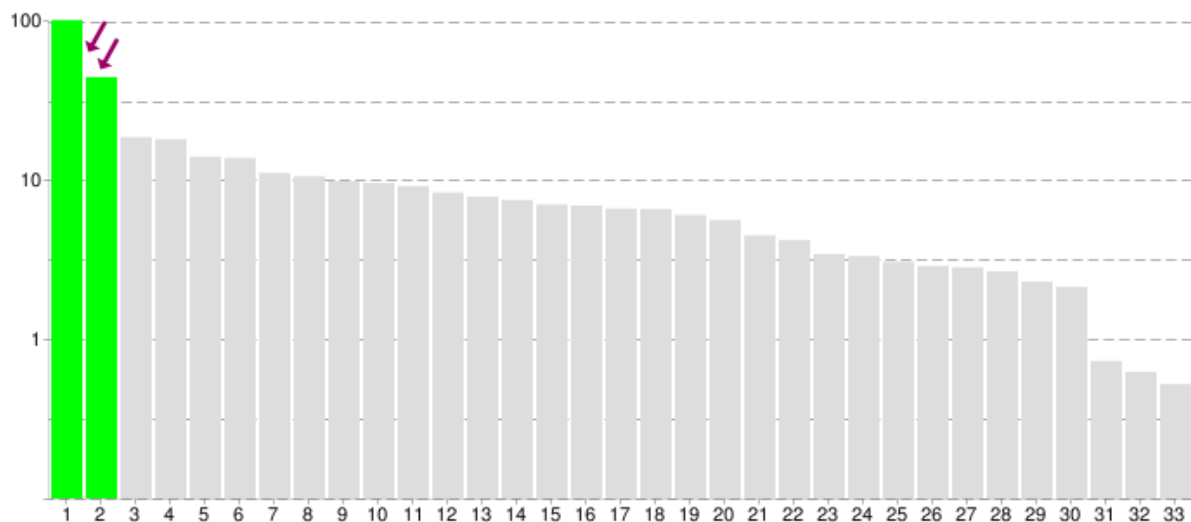


Abbildung 33: Leistung der Objekt-Datenbank ObjectDB (grün) im Vergleich zu anderen JPA/DBMS-Kombinationen (grau)
(<http://www.jpab.org/ObjectDB.html>)

Dokumentenbasierte Datenbanken

Dokumentbasierte Datenbanken, die in Kapitel 2.1.5. vorgestellt wurden, zeichnen sich dadurch aus, dass Daten in Form von Dokumenten abgelegt werden. Da die Daten idealerweise zusammenhangslos sind, können verschiedene Daten unterschiedlichen Typs in ein Dokument gespeichert werden. Dies erleichtert das Speichern im Vergleich zu anderen Datenbankmodellen und dadurch, dass keine feste Struktur vorgegeben ist, werden Schreib- und Leseoperationen performant ausgeführt. Dadurch können auch große Datenmengen sehr performant gelesen und geschrieben werden.

Referenzen bilden die Schwachstelle in einer dokumentenbasierten Datenbank. Um Referenzen in diesem Datenbankmodell abzubilden, würde man das referenzierte Objekt mit in das gleiche Dokument ablegen müssen. In diesem Fall würden im Vergleich zu RDBMS die

Join-Operationen entfallen, jedoch würde beim Lesen eines solchen Objekts der gesamte Inhalt geladen werden, also auch dann, wenn Attribute nicht gebraucht werden. Es ist ebenfalls möglich, Datensätze in mehrere Dokumente zu unterteilen und in den gewünschten Dokumenten mit einer ID auf ein bestimmtes Dokument zu referenzieren, was zu zusätzlichen Abfragen führt (Klößner, 2016).

Im Vergleich zu relationalen Datenbanken, wo Tabellen durch Schreibvorgänge gesperrt werden und der SQL-Befehl warten muss, um eine Tabelle auszulesen, erfolgt das Auslesen von Daten eines Dokuments ohne das Abwarten der Vollendung einer Schreiboperation. Es werden die Daten ausgegeben, die vorhanden sind. Daher ist es wichtig, diese Art der Speicherung nur zu wählen, wenn die Konsistenz der Daten eine zweitrangige Rolle spielt. An dieser Stelle wird die Konsistenz gegen ein schnelles Auslesen der Daten eingetauscht.

Ein weiterer Vorteil gegenüber der RDBMS ist die Skalierung, denn bei RDBMS ist die Verteilung von Tabellen und deren Verknüpfung mit unterschiedlichen Systemen sehr komplex. Lesevorgänge werden unter Verwendung von Replikas weiter verlangsamt, denn bei RDBMS wird der Leseprozess erst bei Vollendung der Transaktion und der Replikation der Datensätze ausgeführt. In dokumentbasierten Systemen werden die Datensätze auf sogenannte Knoten verteilt und nacheinander bei Änderung eines Datensatzes synchronisiert. Lesen ist jederzeit möglich, doch geht man auch hier den Kompromiss ein, Performanz und Einfachheit gegen die Konsistenz der Daten zu tauschen.

Abbildung 34 zeigt ein Dokument welches Datensätze aus zwei Tabellen umfasst.



Abbildung 34: Dokument einer dokumentbasierten Datenbank
(<http://wi-wiki.de/lib/exe/fetch.php?cache=&media=bigdata:documentstore.png>)

4.2 Softwaretuning

Im Kapitel Softwaretuning geht es darum, angewendete Software so zu konfigurieren, dass diese möglichst effizient als Hilfsmittel genutzt werden kann, um letztendlich die Performanz des gesamten Systems zu steigern. In diesem Kapitel werden die im Verborgenen agierende Funktionsweise des ORM-Frameworks Hibernate und die Einstellungen des Microsoft SQL Server kritisch diskutiert. Zudem wird untersucht, was genau mit welchen Einstellungen passiert und wie sich diese auf die Geschwindigkeit des Systems auswirken.

4.2.1 Hibernate Optimierung

Im Folgenden sind Funktionen, Arbeitsweisen und Methoden von Hibernate jeweils in Unterkapitel gegliedert. Neben diesen Unterkapiteln existieren weitere Aspekte, an denen das Framework Hibernate hinsichtlich seiner Performanz verbessert werden kann. Zu beachten ist jedoch, dass dies eine Einstellungssache ist und nur dann einen Performanzgewinn mit sich bringt, wenn diese an das System angepasst sind.

Optimistic Locking

Viele parallellaufende und vor allem lange Transaktionen über mehrere Tabellen hinweg führen in absehbarer Zeit dazu, dass Datenquellen für andere Transaktionen gesperrt werden. Um die Lesegeschwindigkeit zu erhöhen und die Wartezeit zu verringern, muss versucht werden, diese Sperrungen der Datenquellen so gering wie möglich zu halten.

Optimistic-Locking ist eine Strategie zum Umgang mit der Sperrung der Daten. Diese Methode erlaubt die Durchführung mehrerer Transaktionen in einem System, ohne deren Datenquelle zu sperren. Jedoch wird vor jedem *commit* der Transaktion überprüft, ob Modifikationen durch andere Transaktionen stattgefunden haben. In diesem Fall wird ein Rollback veranlasst, um die Daten auf den Ursprungszustand zu setzen. Die Methode des Optimistic-Locking eignet sich besonders gut für Systeme, die ihre Entitäten häufig ablesen müssen. Für Systeme, die ihre Daten oft bearbeiten, ist diese Methode jedoch weniger gut geeignet.

In Hibernate gibt es zwei Arten, die Methode des Optimistic-Locking umzusetzen. Zum einen mittels einer Art *Versionierung* und zum anderen mittels eines *Zeitstempels*. Im ersten Fall gilt, wenn die Anwendung eine Transaktion ausführt, wird eine Versionsnummer gespeichert. Wenn eine weitere Transaktion diese Datenquelle beansprucht und eine Änderung an ihr vornehmen will, würde erst ein Vergleich der Versionen stattfinden, und nachdem ein Benutzer als erster *commit* angegeben hat, würden alle folgenden Benutzer benachrichtigt, dass die Datei für einen Schreibzugriff gesperrt ist und eine Bearbeitung nicht stattfinden kann. Die Benutzung des Zeitstempels ist weniger zuverlässig als die eben erläuterte Variante, erfolgt jedoch nach dem gleichen Prinzip, nur werden in diesem Fall die Zeitstempel (engl. timestamp) miteinander verglichen. Abbildung 35 zeigt einen Code-Ausschnitt mit der Versionierung und in Abbildung 36 ist die Variante mit dem Zeitstempel zu sehen.

```
@Version
@Column(name="OPTLOCK")
Public Integer getVersion() { ... }
```

Abbildung 35: Optimistic-Locking mittels Version

```
@Version
public Date getLastUpdate() { ... }
```

Abbildung 36: Optimistic-Locking mittels Timestamp

Im Gegensatz dazu existiert die Methode des Pessimistic Locking, hier wird davon ausgegangen, dass Transaktionen gegenseitig in einen Konflikt geraten werden. Das erfordert, dass

Daten gesperrt werden und die Sperrung erst aufgehoben wird, wenn die Daten nicht mehr genutzt werden (Hibernate Community Documentation, n.d.).

AllocationSize()

Um IDs für Entitäten automatisch zu generieren, wird die Annotation `@GeneratedValue()` verwendet. Jedoch hat diese einen beachtlichen Nachteil, denn jedes Mal, wenn eine Entität in eine Datenbank gespeichert werden soll, findet ein Lesezugriff auf die Datenbank statt, um die ID zu bestimmen. Anschließend wird ein weiterer Zugriff auf die Datenbank stattfinden, in dem die Entität in die Datenbank abgebildet wird. Bei der Erstellung vieler Entitäten ist diese Art der Generierung weniger performant.

Um die Performanz der Zugriffe zu maximieren, gibt es den Parameter `allocationSize`, dem wie in Abbildung 37 ein Wert-N zugesprochen wird. Dabei ist N durch einen Integer-Wert zu ersetzen. Dieser Parameter besagt, dass der Lesezugriff auf die DB, um die ID zu bestimmen, nur alle N male geschehen soll. Das heißt, wenn eine Entität in die Datenbank gespeichert werden soll, findet eine Anfrage an die Datenbank statt, um die nächste zu benutzende ID zu ermitteln (`next_val`) und die neue Entität mit dieser ID in der Datenbank zu hinterlegen. Die neu ermittelte ID wird lokal gespeichert. Wenn eine weitere Entität in die Datenbank gespeichert werden soll, wird keine Anfrage mehr an die Datenbank zur Bestimmung der ID (`next_val`) geschickt, sondern es wird die lokal gespeicherte ID genutzt, die sich jedes Mal um 1 erhöht, bis der Wert N erreicht wird.

Wenn N der Wert 50 zugeschrieben wird, bedeutet dies, dass 50 Speichervorgänge durchgeführt werden können, ohne eine erneute Anfrage bezüglich der ID an die Datenbank zu senden. In diesem Beispiel wäre also bei 50 Speichervorgängen nur bei dem ersten eine Ermittlung der ID mithilfe der Datenbank nötig. Abstrahiert bedeutet es also, dass auf N-1-Anfragen verzichtet werden kann.

```
@Id
@GeneratedValue (Strategy = GenerationType.SEQUENCE, generator = "MeinIDGenerator")
@SequenceGenerator(name = "MeinIDGenerator ", allocationSize = N
@private int Id;
```

Abbildung 37: AllocationSize
(Wilming, 2013)

Im Falle, dass zwei *EntityManager* Entitäten in eine Datenbank speichern wollen und beide jeweils den `allocationSize`-Parameter nutzen, würde der erste die *ID* = 1 bekommen und der folgende *EntityManager* würde die *ID* = 1 + N bekommen (Roy, 2010)

Eager- vs. Lazyloading

Ein weiterer Punkt, an dem man die Performanz verbessern kann, liegt in den Relationen `@OneToMany` oder `@ManyToOne`. Hier existiert ein `Fetch`-Parameter, dem man einem Wert *eager* bzw. *lazy* zuordnen kann. *Eager* bedeutet, dass alle Referenzen zu diesem Objekt direkt geladen werden, *lazy* wiederum bedeutet, dass die referenzierten Objekte nur bei Bedarf geladen werden (Siehe Abbildung 38). Wenn man das Zugriffsverhalten kennt, kann man entscheiden, wie die Daten geladen werden. In der Regel ist die Einstellung *lazy* die bevorzugte Wahl, da hier die

Daten nur bei Bedarf bezogen werden und somit Ressourcen nicht unnötig verschwendet werden (Wilming, 2013).

```
@ManyToOne(fetch = FetchType.Lazy)
private Artikel artikel;

@ManyToOne(fetch = FetchType.Eager)
private Artikel artikel;
```

Abbildung 38: Syntax Eager- und Lazyloading
(Wilming, 2013)

First- und Second-Level-Cache

Hibernate hat einen First-Level-Cache und einen Second-Level-Cache, um Objekte zwischenspeichern und bei Bedarf Objekte aus dem Cache zu laden, anstatt eine neue Anfrage an die Datenbank zu schicken. Ein Cache dient zur Zwischenspeicherung von Daten und deren schnelleren Abrufung. Abbildung 39 zeigt, wie die Komponenten Datenbank, First-Level-Cache, Sitzungs-Objekt und Client zusammen agieren.

Der First-Level-Cache ist standardmäßig eingeschaltet und kann auch nicht ausgeschaltet werden. Der First-Level-Cache ist mit einer Sitzung (Session), die vom *SessionFactory* erzeugt wird, verbunden. So werden die in dieser Sitzung erzeugten Objekte, die beim ersten Mal von der Datenbank genommen werden, in dem Cache gespeichert und auch wieder gelöscht, sobald die Sitzung terminiert wird. Bei einem zweiten Aufruf dieses Objekts wird dieser aus dem Cache und nicht aus der Datenbank geladen. Zudem ist ein Objekt einer Sitzung nicht außerhalb dieser Sitzung aufrufbar. Mit der Methode *evict()* können einzelne gezielte Objekte aus dem Cache gelöscht werden. Die Methode *clear()* sorgt dafür, dass der ganze Cache geleert wird. Diese Methoden werden genutzt, um den Cache zu entlasten und *Out-of-Memory*-Fehler zu unterbinden (Gupta, 2013))

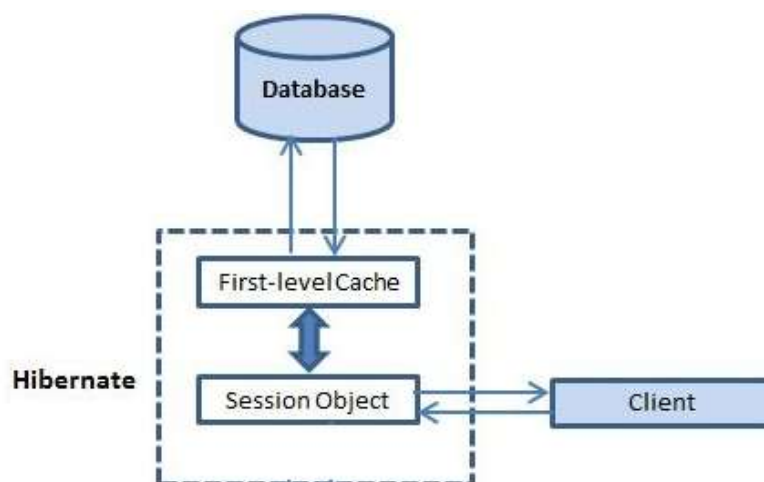


Abbildung 39: First-Level-Cache, Session-Objekt
(https://cdn2.howtodoinjava.com/wp-content/uploads/hibernate_first_level_cache.jpg)

Der Second-Level-Cache ist im Gegensatz zum First-Level-Cache nicht nur in ein und derselben Sitzung verwendbar, sondern in allen Sitzungen, die von derselben *SessionFactory* erzeugt wurden. In diesem Fall wird der Cache erst dann gelöscht, wenn die *SessionFactory* terminiert

wird und nicht bei einzelnen Sessions. Der Second-Level-Cache ist nicht standardmäßig eingeschaltet (enabled).

Der Second-Level-Cache wird eingesetzt, wenn ein Objekt aufgerufen werden soll und nicht im First-Level-Cache vorhanden ist. Dann wird geprüft, ob es im Second-Level-Cache vorhanden ist. Falls in beiden Caches kein Objekt gefunden wurde, wird das geforderte Objekt von der Datenbank bezogen. Wenn das gewünschte Objekt jedoch im Second-Level-Cache und nicht im First-Level-Cache gefunden wird, wird dieses Objekt aus dem Second-Level-Cache geladen. Allerdings wird, bevor es geladen wird, eine Kopie des Objekts zuerst in den First-Level-Cache gespeichert, sodass bei einem erneuten Aufruf dieses Objekts nur im First- und nicht noch weiter im Second-Level-Cache gesucht werden muss.

Wenn eine direkte Änderung auf der Datenbankseite erfolgt, wird sich das Objekt im Second-Level-Cache nicht ändern. Für diesen Fall könnte man ein Kriterium „timeToLiveSeconds“ integrieren, das den Cache nach Ablauf der eingestellten Zeit für ungültig erklärt. Im Anschluss würde Hibernate diesen Cache wieder neu aufbauen und die Daten wären wieder synchronisiert.

Abbildung 40 zeigt die Zusammenarbeit von der Datenbank (Database), Client und Hibernate sowie das Zusammenspiel von First- und Second-Level-Cache und der *SessionFactory* innerhalb von Hibernate (Gupta, 2013).

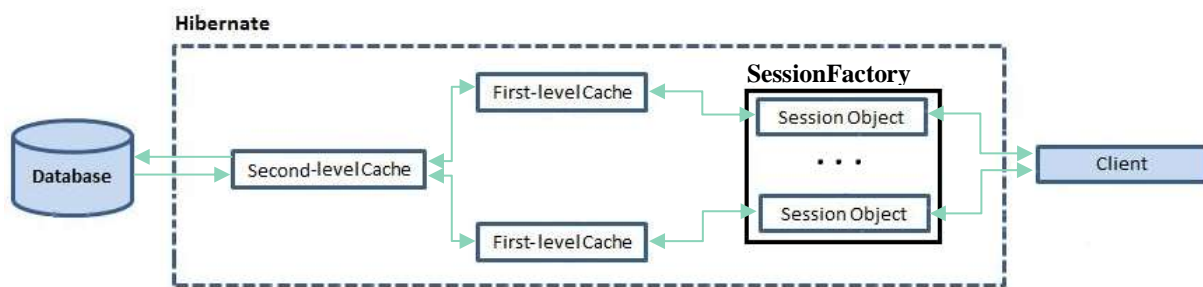


Abbildung 40: Second-Level-Cache (eigene Darstellung, in Anlehnung an Abbildung 42)

Paginierung

Hibernate bietet die Möglichkeit der Paginierung. Mit Paginierung ist es möglich, große Datenmengen „häppchenweise“ abzurufen, denn die Verarbeitung kleinerer Datenmengen erfolgt in der Regel schneller als die Verarbeitung großer Datenmengen. Darüber hinaus kann es vorkommen, dass nur bestimmte Datenblöcke gebraucht werden, zum Beispiel nur die ersten 50 Kundennamen. Um dann nicht mehr Daten abzurufen, als gebraucht werden, liefert Hibernate die beiden Methoden *setFirstResult(n)* und *setMaxResults(m)*, wobei *n* und *m* Integer-Werte sind.

Die Methode *setMaxResults(m)* besagt, dass nur *m*-Datensätze zurückgegeben werden. Die Methode *setFirstResult(n)* bedeutet, dass die ersten *n*-Datensätze nicht zurückgegeben werden (Mihalcea, 2019).

Abrufen mehrere Entitäten

Bis zur Hibernate-Version 5.1 gab es keine elegante Methode, viele Entitäten von der Datenbank mittels ID abzurufen. Es gab die Option, die *Find*-Methode vom *EntityManager* für jeden primären Schlüssel aufzurufen. Jedoch benötigt dies für jede Anfrage eine Verbindung

zur Datenbank, was für eine kleine Anzahl an IDs tragbar ist, für eine große Anzahl aber zu performanzlastig und die Applikation verlangsamt.

Die zweite Option bestand darin, dass eine Query generiert wurde, in der die primären Schlüssel der Entitäten als Menge angegeben wurden. Dieser Prozess verlief durch den Mengen-Operator `IN`. Abbildung 41 veranschaulicht die zweite Methode (Jansen, Thoughts On Java, 2019).

```
List<Long> ids = Arrays.asList(new Long[]{1L, 2L, 3L});
List<PersonEntity> persons = em.createQuery
    (
        "SELECT p FROM Person p
         WHERE p.id IN :ids"
    ).setParameter("ids", ids).getResultList();
```

Abbildung 41: Fetch vor Hibernate 5.1
(Jansen, Thoughts On Java, 2019)

Einige Datenbanksysteme, Oracle mitinbegriffen, haben eine begrenzte Anzahl an Parametern innerhalb der `IN`-Klausel und aus diesem Grund ist die Methode für solche Datenbanksysteme nicht geeignet. Zusätzlich können Performanzprobleme auftreten, da alle Daten in einem Satz bezogen werden. Ein weiterer Nachteil ist, dass nicht im First-Level-Cache nach bereits gespeicherten Entitäten gesucht wird. Um diese Probleme zu beheben, müsste zusätzlichen Code produziert werden, was wiederum die Komplexität erhöhen würde.

Mit Hibernate 5.1 wurde eine Api veröffentlicht, die es erlaubt, mit einem Aufruf dieser Schnittstelle mehrere Entitäten zu laden und durch die zusätzlichen Funktionen die genannten Nachteile zu beseitigen.

Mit dem Aufruf der Methode `byMultipleIds(Class entityClass)` auf die Hibernate-Sitzung und mit `multiLoad(Params ...)` kann eine große Menge Daten geladen werden.

In diesem Fall erstellt Hibernate eine mit `IN` eingeleitete Mengenoperation, die die primären Schlüssel beinhaltet. Abgesehen von der Nutzung der neuen Methoden ist im Vergleich zu Abbildung 41 auf den ersten Blick kein Unterschied gegeben. An dieser Stelle gilt es zu beachten, dass, wenn man Hibernate in Kombination mit JPA nutzt, in der Regel der `EntityManager` genutzt wird, um Anfragen an die Datenbank zu senden. Damit die neuen Methoden genutzt werden können, wird die Hibernate-Session benötigt. Eine Hibernate-Session bekommt man mit dem Aufruf der `unwrap`-Methode nach folgendem Schema: `Session session = em.unwrap(Session.class)`

Abbildung 42 zeigt einen Code-Ausschnitt mit der Syntax der neuen Schnittstelle.

```
MultiIdentifierLoadAccess<PersonEntity> multiLoadAccess =
    session.byMultipleIds(PersonEntity.class);
List<PersonEntity> persons = multiLoadAccess.multiLoad(1L, 2L, 3L);
```

Abbildung 42: Fetch nach Hibernate 5.1
(Jansen, Thoughts On Java, 2019)

Weitere Methoden wie `withBatchSize()` sorgen für den Unterschied zu Abbildung 41, denn diese erlaubt es, die Größe eines Batch zu bestimmen. In Fällen, in denen die Anzahl an zurückgelieferten Entitäten eine große Menge darstellt, ist es sinnvoll, diese in mehrere Batches zu unterteilen. So könnte man beispielsweise den First-Level-Cache nach dem ersten Batch entlasten, bevor der zweite geladen wird.

Ebenfalls besteht durch das `MultiIdentifierLoadAccess`-Interface die Möglichkeit, Hibernate mitzuteilen, dass der First-Level-Cache vor einer Anfrage an die Datenbank nach bereits geladenen Entitäten durchsucht werden soll (Abbildung 43).

```
PersonEntity p = em.find(PersonEntity.class, 1L);
log.info("Fetched PersonEntity with id 1");
Session session = em.unwrap(Session.class);
List<PersonEntity> persons =
    session.byMultipleIds(PersonEntity.class).enableSessionCheck(true)
        .multiLoad(1L, 2L, 3L);
```

Abbildung 43: Daten aus First-Level-Cache lesen, vor Datenbankzugriff
(Jansen, Thoughts On Java, 2019)

Entitäten ohne Assoziation verbinden

Bis Version 5.0 war es ohne Weiteres nicht möglich, Entitäten, die keiner Relation zueinander zugewiesen wurden, mittels eines *Join*-Operators miteinander zu verbinden.

Die einzige Möglichkeit bestand darin, ein Query zu erzeugen und in der *From*-Klausel beide Entitäten, die man verbinden möchte, anzugeben. Dies hatte zur Folge, dass daraus ein kartesisches Produkt entstand. Um dessen Ausgabe zu reduzieren, wurden in der *Where*-Kausel entsprechende Kriterien angegeben (Jansen, Thoughts On Java, 2019).

In Abbildung 44 ist ein Code-Beispiel dargestellt.

```
List<Object[]> results = em.createQuery("SELECT p.firstName,
p.lastName, n.phoneNumber FROM Person p, PhoneBookEntry n
WHERE p.firstName = n.firstName AND p.lastName =
n.lastName").getResultList();
```

Abbildung 44: Zwei Entitäten ohne Relation mittels kartesischen Produkts verbinden
(Jansen, Thoughts On Java, 2019)

Diese sogenannten *Cross-Join* sind schwer zu lesen und kosten die Datenbank viel mehr Ressourcen als ein *Inner-Join*. Darüber hinaus gab es keine Möglichkeit, einen *Outer-Join* zu nutzen.

Mit der Hibernate-Version 5.1 wurden explizite Joins für Entitäten ohne Assoziationen eingeführt. Die Syntax ist der Datenbanksprache SQL, wie man in Abbildung 45 (*Inner-Join*) sehen kann, sehr ähnlich. Anstatt nun beide Entitäten in der *from*-Klausel stehen und ein kartesisches Produkt bilden, wird das zu verbindende Objekt mit dem *Join*-Operator referenziert und dessen Kriterien für die Filterung in der *On*-Klausel gesetzt. Dies wird von Hibernate zu einem *Inner-Join* umgewandelt. Ein *Outer-Join* wird erstellt, indem einem Join das Präfix *left* hinzugefügt wird.

```
List<Object[]> results = em.createQuery("SELECT p.firstName,
p.lastName, n.phoneNumber
FROM Person p JOIN PhoneBookEntry n ON p.firstName = n.firstName
AND p.lastName = n.lastName").getResultList();
```

Abbildung 45: Zwei Entitäten ohne Relation mittels Join-Operator verbinden
(Jansen, Thoughts On Java, 2019)

N+1-Problem

Das N+1-Problem beschreibt Anfragen an eine Datenbank der objektrelationalen Abbildung, die nach einem bestimmten Muster erfolgen. In diesem Muster werden genau N+1-Anfragen durch ein *Select*-Statement an eine Datenbank geschickt.

In einem Szenario, indem beispielsweise viele Schüler-Entitäten in einer Many-To-One-Relation zur Entität *Klasse* stehen und man alle Daten der Schüler einer Klasse ausgeben möchte, würde eine typische Query wie in Abbildung 46 aussehen.

```
Select * from Klasse;  
Select * from Schüler where Schüler.schüler_id = Klasse.schüler_id
```

Abbildung 46: Select-Anfragen N+1

Es würde eine Anfrage an die Datenbank geschickt werden, um das Eltern-Objekt zu bekommen und jeweils eine weitere Query für jedes Kind-Objekt. Somit werden mehr Anfragen an die Datenbank gestellt, als eigentlich benötigt werden und infolgedessen werden mehr Ressourcen verbraucht. Dies ist auf das Ladeverhalten der Entitäten zurückzuführen. Wie im Absatz *Eager- vs. Lazyloading* beschrieben, ist *lazy* die Standardeinstellung, somit werden nur Objekte geladen, die gebraucht werden. Zudem wird das N+1-Problem nicht immer erkannt, da dies durch Hibernate zum Teil maskiert wird. Hibernate bearbeitet das OR-Mapping und für den Benutzer ist zunächst nicht sichtbar, wie Hibernate agiert. Der Benutzer kann den durch Hibernate erstellten SQL-Code untersuchen, doch ist das mit Zeitaufwand verbunden. Somit vertraut der Benutzer darauf, dass Hibernate die Anfragen richtig bearbeitet. Durch das Aktivieren von SQL-Logging, das die Übersetzung der HQL-Befehle in SQL anzeigt, wird erkennbar, dass beispielsweise eine Anfrage N+1 Selects an eine Datenbank sendet.

Wurde dieses Problem erkannt, ist die Lösung ein HQL-*Fetch-Join*. Dieser sorgt dafür, dass die Entitäten in einem *Left-Outer-Join* zusammengefügt werden und alle Informationen in einem einzigen Befehl geladen werden.

Abbildung 47 zeigt, wie der in HQL genutzte *Fetch-Join* durch Hibernate in SQL umgeschrieben wird.

```
HQL: Select * from Klasse k join fetch k.employees Schüler  
SQL: SELECT * FROM Klasse k LEFT OUTER JOIN Schüler s  
      ON k.klassenId = s.klassenId
```

Abbildung 47: HQL Fetch-Join und wie diese in SQL interpretiert wird

Auch ist es mittels Query-Kriterien möglich, das Ladeverhalten der Objekte auf *eager* zu setzen. Somit werden die Objekte mit allen Informationen in einem Statement direkt geladen. Kriterien könnten, wie in Abbildung 48 angegeben werden (Suda, 2014).

```
Criteria criteria = session.createCriteria(Department.class);  
criteria.setFetchMode("employees", FetchMode.EAGER);
```

**Abbildung 48: Lösung des N+1-Problems mittels Kriterien a) und Fetch-Join b)
(Suda, 2014)**

4.2.2 SQL Server-Konfiguration

Database Engine Tuning Advisor (DTA)

Der DTA analysiert Datenbanken und gibt Vorschläge, wie man Anfragen auf geeignete Weise verbessern kann. Der DTA wird verwendet, um nicht performante Queries zu untersuchen und diese anschließend zu optimieren. Auch über mehrere Datenbanken können Sets an Queries verbessert werden. Darüber hinaus kann eine Was-wäre-wenn-Analyse von Queries durchgeführt werden, d. h. untersucht werden, wie eine Query sich verhalten würde, wenn sich das Design der Datenbank ändern würde. Zusätzlich kann DTA auch zur Verwaltung der Speicher genutzt werden.

Da der Tuning-Advisor sehr viele Ressourcen für die Analyse verwenden kann, sollte man diesen nur dann verwenden, wenn der Server frei ist (Microsoft SQL Docs, 2017).

Execution-Plan-Choice-Regression

Der Microsoft SQL Server 2017 (14.x) verfügt über das neu eingebaute Feature Automatic Tuning. Dieses Feature teilt dem Benutzer mit, wenn eine Query verbessert werden kann und um Query welche es sich handelt. Zudem wird vorgeschlagen, wie diese verbessert werden kann. Dieser Vorgang lässt sich durch Aktivierung auch komplett automatisieren. Zwar bietet der SQL Server-Database-Engine detaillierte Einsicht in die Queries und Indizes, die man überwachen muss, doch ist deren eigenhändige Untersuchung zeitraubend. Man kann Arbeiten an die Database-Engine delegieren, die automatische Features nutzt, um die Performanz zu erhöhen.

Die automatisierte Verbesserung besteht aus einem kontinuierlichen Prozess der Überwachung und der Analyse. So verstehen die automatischen Features die Eigenschaften des Workloads und können darauf basierend Verbesserungen vorschlagen. Als erstes wird jede Veränderung, die automatisiert im System vollzogen wird, aufgezeichnet und die Performanz des Systems wird berechnet. Der Vergleich beider Workloads stellt sicher, ob eine Verbesserung stattgefunden hat. Diese Aktion wird jedoch rückgängig gemacht, wenn die Performanz nicht steigen sollte. Die Validierung der Änderungen erfolgt durch die Execution-Plan-Choice-Regression-Methode.

Der SQL Server-Database-Engine kann unterschiedliche Ausführungspläne für Transaktionsqueries nutzen. Wie gut ein Ausführungsplan ist, hängt von Statistiken, den Indizes und weiteren Faktoren ab. So ist es möglich, dass sich die Ausführungspläne mit der Zeit ändern. Dabei wird der letzte Gute Plan verwendet, falls die Änderung eines Plans nicht zu einer Steigerung der Performanz geführt hat (Abb. 49). Dabei gilt der *gute Plan* als der Plan, der bis dato die beste Performanz aufwies.

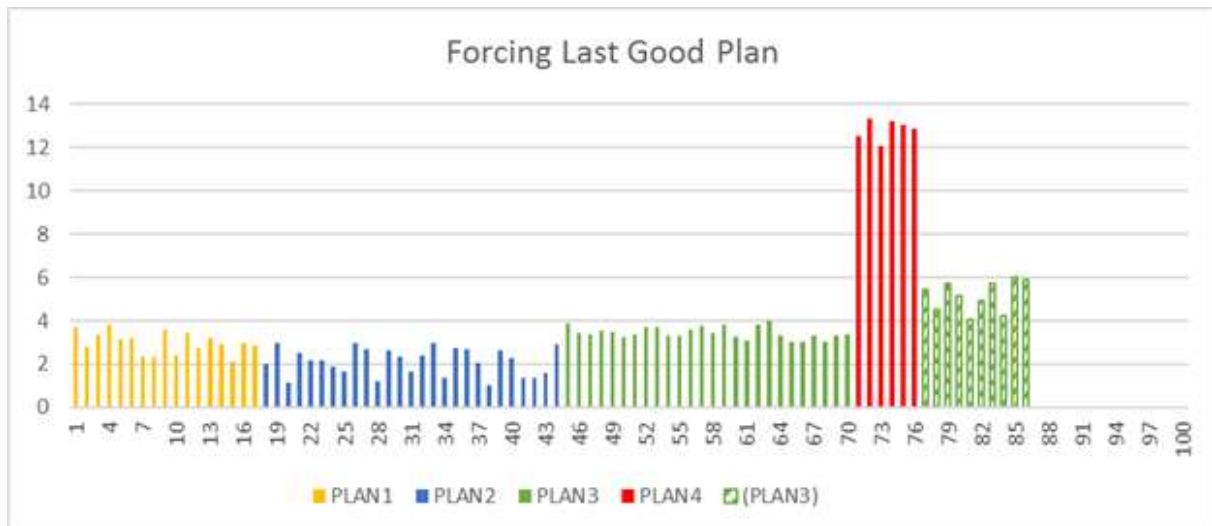


Abbildung 49: Letzten guten Plan forcieren

(<https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/media/force-last-good-plan.png?view=sql-server-2017>)

Ohne das automatisierte Verbessern müsste der Datenbank-Administrator in periodischen Zeitabständen das System überwachen und nach Regressionen Ausschau halten. Mit der Prozedur *sp_query_store_force_plan* müsste der Administrator einen *guten Plan* aus der Vergangenheit auswählen und anwenden. Um sicherzustellen, dass ein Performanzgewinn stattgefunden hat, sollte vor- und nach der Einstellung des *guten Plans* die Performanz gemessen werden. In der Regel sollte der neuste *gute Plan* ausgewählt werden, da ältere Pläne überholt sein könnten und möglicherweise mit dem jetzigen System nicht mehr kooperieren. Ebenfalls sollte ein solcher Plan nicht für immer erzwungen werden, um der Database-Engine die Chance zu geben, weitere, eventuell bessere Pläne zu generieren. Im MSSQL Server 2016 (13.x) kann man den Plan im Query-Store mittels System-Views anschauen. Ab der Version 2017 (14.x) ist der Database-Engine dafür zuständig, Regressionen zu entdecken, anzuzeigen und Verbesserungen vorzuschlagen (Microsoft SQL Docs, 2017).

Paralleles Ausführen erzwingen

Bei der Parallelisierung werden größere Aufgaben in mehrere kleine Aufgaben unterteilt und gleichzeitig abgearbeitet, anschließend werden alle Einzelergebnisse kombiniert und zum Endergebnis zusammengefügt.

Um eine echte Parallelisierung zu ermöglichen, muss der SQL Server auf einem Server laufen, der mehrere Prozessoren hat. So kann eine einzelne Aufgabe von einem Prozessor übernommen werden, was dazu führt, dass mehrere Prozessoren tatsächlich parallel an ihren Aufgaben arbeiten. Im SQL Server gibt es einen Schwellwert, der standardmäßig auf 5 gesetzt ist und angibt, wann bzw. ob eine Abfrage parallelisiert werden soll. Dabei stellt die Zahl einen abstrahierten Kostenwert dar. Ist dieser Schwellwert auf 1 gesetzt, wird immer eine serielle Anfrage und niemals eine parallele durchgeführt. Eine Anfrage wird demnach parallelisiert, wenn die Kosten den Schwellwert übersteigen und der parallele Plan performanter ist als der serielle (Microsoft SQL Docs, 2017).

In manchen Fällen werden Anfragen seriell und nicht parallel durchgeführt, weil die Anfragen beispielsweise relationale Operatoren enthalten und somit nicht parallel laufen können. Ein weiterer Grund kann der Kostenfaktor sein. Ebenfalls kann der Query-Optimizer wegen marginaler Verbesserungen serielle Ausführung gegenüber der parallelen bevorzugen. Falsche Entscheidungen können aber auch durch die Einschränkungen entstehen, die der Optimizer hat, oder durch falsche Informationen, die dieser bekommt.

Mit dem erzwingen eines parallelen Plans, kann ein Performanzgewinn herbeigeführt werden. Dies wird möglich gemacht indem man die auszuführende Anfrage um die *Option*-Klausel mit dem Parameter *QUERYTRACEON 8649* erweitert und somit das Trace Flag 8649 aktiviert.

Auch kann man in den SQL Server-Versionen 2016 und neuer die *Option*-Klausel mit (*Use HINT("ENABLE_PARALLEL_PLAN_PREFERENCE")*) verwenden, diese wurde eingeführt, um die *QUERYTRACEON* Option abzulösen. Diese Methode kann im Gegensatz zu der eben vorgestellten auch ohne Adminrechte ausgeführt werden.

In Tabelle 13 sind die Anfragedauer der seriellen und der parallelen Option zu sehen (Yaseen, MSSQLTips, 2017).

Methode	CPU-Zeit	Vergangene Zeit
Seriell	62 ms	71 ms
<i>Option(QUERYTRACEON 8649)</i>	92 ms	43 ms
<i>Option (Use HINT("ENABLE_PARALLEL_PLAN_PREFERENCE"))</i>	95 ms	39 ms

Tabelle 13: Parallele Ausführung vs. serielle Ausführung
(Yaseen, MSSQLTips, 2017)

5 Archivierungsverfahren

Als ein Archiv wird ein Ort bezeichnet, der zur Aufbewahrung von Objekten genutzt wird. Objekte, die in einem Archiv gelagert werden, können unterschiedliche Formen haben, beispielsweise können es Schriftstücke, Bilder, Videos oder Pläne sein. Die Funktion eines Archivs ist es, diese Objekte so aufzubewahren, dass sie auch über eine lange Zeit erhalten bleiben. Zudem muss es möglich sein, dem Archiv weitere Objekte hinzuzufügen und auch einen Zugang geben, über den Inhalte der Objekte gelesen werden können (Stadtarchiv-Lemgo.de, n.d.).

In der digitalen Welt kann sich das Aufbewahren von Objekten in einem Archiv jedoch als kompliziert erweisen. Zum einen können die zu speichernden Objekte komplex sein und zum anderen können für bestimmte Archive unterschiedliche Anforderungen bestehen. Durch die Auslagerung von Archivdaten, die in der Regel 80 % der gesamten Daten betragen, wird ein schnellerer Zugriff auf die restlichen Daten gewährt. Weil es heutzutage wichtig ist, Daten mittelfristig bzw. langfristig zu speichern, kommt es vor, dass die Menge der Daten, die den Speicherplatz belegen, so groß ist, dass Systeme damit nicht umgehen können und langsamer werden. Wegen der Anforderung, dass ein Archiv zu jeder Zeit zugänglich sein muss, ist es nicht möglich, ohne Weiteres Daten aufzuteilen.

Angesichts der immer weiterwachsenden Anforderungen an ein Archiv und der Erhöhung ihrer Komplexität wurden Archivierungsverfahren ausgearbeitet.

In einem System wird zwischen heißen Daten (Hot Data) und kalten Daten (Cold Data) unterschieden. Als heiße Daten werden die Daten bezeichnet, die im täglichen Betrieb verwendet werden. Kalte Daten wiederum sind Daten, die für gewöhnlich in einem Archiv untergebracht werden, da sie selten bis nie aufgerufen werden.

Aus dieser Aufteilung in heiße und kalte Daten ergeben sich einige bedeutende Vorteile: Da heiße Daten in der Regel nur 20 % der gesamten Daten darstellen, erfolgen Zugriffe schnell. Darüber hinaus können die wenigen heißen Daten auf schnelle Speichermedien, wie etwa Solid State Drive (SSD), die mit wachsender Speicherkapazität einen exponentiellen Kostenzuwachs aufweisen, gespeichert werden. Diese Speichermedien sind optimiert und können Lese- und Schreiboperationen schnell und effizient umsetzen. Im Gegensatz dazu können kalte Daten, die keine schnellen Lese- und Schreiboperationen erfordern, auf kostengünstige High Disk Drive (HDD) zurückgreifen. HDD-Speichermedien sind zwar im Vergleich langsamer, weisen jedoch eine signifikant höhere Speicherkapazität auf.

Durch das Aufteilen der Daten werden auch die Back-ups kleiner. Mit einem Back-up werden Schnappschüsse von einem System erstellt und zur Datensicherung abgespeichert. Back-ups können abhängig von der Menge der Daten unterschiedlich groß werden und werden oft überschrieben und aktualisiert. Da die heißen Daten nur 20 % aller Daten ausmachen, fallen die Back-ups dementsprechend kleiner aus. Dadurch werden sie schneller und performanter erstellt und zügiger eingespielt, wenn ein Back-up gebraucht wird. Es wird versucht, immer so zeitnah wie möglich Back-ups zu sichern, um bei einem Systemausfall eine möglichst aktuelle Kopie der Daten zu haben (Rouse, 2018).

Das Kapitel beginnt mit der logischen Archivierung, die Daten mittels Hilfsattributen in heiße und kalte Daten aufteilt. In Kapitel 5.1.2 wird beschrieben, wie Daten in unterschiedliche Tabellen separiert werden, wobei jeder Tabelle eine weitere Tabelle mit dem Suffix *_archiv* zugeordnet ist. Des Weiteren werden Daten auch in verschiedenen Datenbanken als Produktiv- und Archivdaten gespeichert.

5.1.1 Logische Archivierung

Bei der logischen Archivierung werden die Daten nicht separiert, sondern bleiben dort, wo sie gespeichert wurden. Zusätzlich wird durch eine Hilfsspalte *istArchiviert*, die bevorzugt vom Typ *Boolean* sein sollte und die Werte *true* oder *false* einnehmen kann, angezeigt, ob ein Datensatz archiviert ist oder nicht.

Eine Anfrage, die die logische Archivierung ausnutzen möchte, muss diese Hilfsspalte als Kriterium im Query beinhalten. Mit dem Kriterium, das beispielsweise *false* sein kann und damit aussagt, dass man nur aktuelle Daten beziehen möchte, werden in einem Scan 80 % der Daten (kalte Daten) übersprungen, was zu einer schnelleren Rückgabe führt.

Nach der gleichen Methode ist es möglich, Daten logisch zu löschen, denn wenn man den Speicher besitzt und die Option hat, Daten nicht löschen zu müssen, kann von dem *Delete*-Operator abgesehen werden. Der *Delete*-Operator nimmt im Vergleich zum *Insert*-Operator mehr Leistung in Anspruch und kann zudem gegebenenfalls Fremdschlüssel-Beziehungen verletzen.

Um diese Methode nutzen zu können, muss Wissen über die Hilfstabelle vorhanden sein. Eine Abfrage ohne dieses Wissen würde über alle Daten laufen, archivierte und aktuelle Daten als Ziel haben und somit maximale Zeit in Anspruch nehmen. Im Vergleich zur Separierung ist die Umsetzung dieser Methode einfach, da nur eine Hilfsspalte hinzugefügt werden muss und die unterliegende Struktur der Daten mit ihren Relationen unangetastet bleiben.

5.1.2 Eine Datenbank mit zusätzlichen Archivtabellen

Bei der Archivierung mit zusätzlichen Archivtabellen werden alle vorhandenen Daten auf eine Datenbank mit zusätzlichen Hilfstabellen gespeichert. Die Hilfstabellen dienen der Archivierung und können deshalb mit dem Suffix *_archiviert* versehen werden. Anschließend werden die zu archivierenden Daten in die Hilfstabelle verschoben.

Große Tabellen, die viele Lese- bzw. Schreibbefehle erhalten, werden durch die Auslagerung solcher Daten in die Hilfstabelle entlastet, wodurch die Ergebnisse schneller zurückgegeben werden. In der Regel sind die aktuellen Daten von Belang und werden fast ausschließlich aus einer Datenbank bezogen.

Sollen jedoch archivierte Daten abgerufen werden, muss man wissen, dass diese in andere Tabellen ausgelagert wurden, um auf sie zugreifen zu können. Sollen jedoch archivierte und aktuelle Daten abgerufen werden, können die Inhalte dieser beiden Tabellen mit dem *Union*-Operator zusammengefügt werden und eine große Tabelle generieren, die sowohl heiße als auch kalte Daten enthält. Dies ist möglich, weil beide Tabellen die gleichen horizontalen Spalten besitzen.

5.1.3 Nutzung von Datenbanken für Produktiv- und Archivdaten

Diese Methode ist vom Prinzip her ähnlich, wie die in Kapitel 5.1.2 vorgestellte Methode. Der Unterschied besteht darin, dass die Daten nicht in zwei verschiedenen Tabellen archiviert werden, sondern es für kalte und heiße Daten eine eigene Datenbank innerhalb einer Serverinstanz gibt. Abgesehen davon, dass die Daten nun verteilt sind und die Datenbank mit den Produktivdaten entlastet wurde, was sich in einer schnelleren Antwortzeit widerspiegelt, können in diesem Fall die Daten auf unterschiedlichen Speichermedien gespeichert werden. So kann die Datenbank mit den Produktivdaten, die schnellen Zugang erfordern, auf einer SSD-Platte und die Datenbank mit den Archivdaten, die keinen schnellen Zugang zu den Daten erfordert, auf einer im Verhältnis kostengünstigen und eher langsamen, aber mit viel mehr Speicher ausgestatteten HDD gespeichert werden.

Da die Daten miteinander mit Relationen verbunden sind, ist es nicht möglich, ohne Weiteres Daten in ein Archiv zu verschieben, ohne die Beziehungen zwischen den Daten zu verlieren. Angenommen, es besteht die Beziehung Klasse zu Schüler, wobei es viele Schüler in einer Klasse geben kann, dann wäre dies eine One-To-Many-Assoziation. Würde ein Klassenobjekt, ohne sich um die Relation zu kümmern, archiviert werden, würde die Relation von Klasse zu Schüler verloren gehen. Dies hätte zur Folge, dass sie bei einem erneuten Aufruf des archivierten Klassenobjekts nicht mehr dem Schüler zugeordnet werden könnte, diese Information wäre somit verloren.

Ein weiteres Problem besteht in der performanten Beschaffung der Daten, denn es soll möglich sein, Daten aus der Produktionsdatenbank, dem Archiv und der Kombination von Archiv und Produktionsdatenbank beschaffen zu können, jedoch ohne das Wissen darüber, wie die Daten gespeichert wurden (Anforderung/F2011/). Dies führt dazu, dass die zwei Datenbanken zu einem Verbund geschlossen werden müssen, sodass sie dem Benutzer als eine logische Einheit präsentiert werden.

Damit die Abhängigkeiten zwischen Tabellen nicht verloren gehen, müssen auch deren Relationen mit in die Archivdatenbank gespeichert werden. Dies erfordert die gleiche Datenstruktur auf beiden Datenbanken. Zudem müssen die Datenbanken von der Struktur her synchron gehalten werden. Das heißt, dass wenn in der Produktionsdatenbank einer Tabelle eine weitere Spalte zugefügt wird bzw. eine Spalte gelöscht wird, muss das gleiche auch in der Archivdatenbank geschehen. Ebenfalls ist beim Kopieren der gegebenen Struktur in die Archivdatenbank darauf zu achten, dass genau die gleichen Primär- und Fremdschlüssel gesetzt werden. Eine Möglichkeit, das zu realisieren, wäre es, eine Hilfstabelle anzulegen, die alle Schlüssel enthält. Die Struktur und die Daten würden in die Archivtabelle kopiert und im Anschluss mithilfe der angelegten Hilfstabelle die Schlüssel dementsprechend gesetzt (Stackoverflow, 2010).

Um Daten aus einer Datenbank, sei es der Produktivdatenbank oder der Archivdatenbank, zu beschaffen, werden hier zwei Vorgehensweisen angesprochen. Zum einen ist es möglich, ein gewisses Kriterium zu integrieren, was die Art der Daten betitelt, die abgerufen werden sollen. Damit wird eine Anfrage direkt an die entsprechende Datenbank geschickt.

Sollen Daten von beiden Datenbanken abgerufen werden, muss ein Kriterium existieren, und zwar entweder eine explizite Angabe oder das komplette Weglassen des Kriteriums, um die Art der Anfrage zu klassifizieren. In diesem Fall wird eine logische Einheit wie im Kapitel 5.1.2, unter Verwendung des *Union*-Operator simuliert, um von der Art her gleiche Tabellen miteinander horizontal zu verbinden. Mit der *gleichen Art* werden Tabellen betitelt, die den gleichen Aufbau und die gleiche Struktur besitzen, wobei deren Inhalt aus archivierten Daten und Produktivdaten besteht. Abbildung 50 zeigt eine *Select*-Anfrage, wie mit einem *Union*-Operator Datensätze aus zwei Tabellen, die auf zwei unterschiedlichen Datenbanken liegen, zusammengeführt werden.

```
SELECT PersonID, FirstName, LastName FROM DB1.dbo.persons
UNION
SELECT PersonID, FirstName, LastName FROM DB2.dbo.persons
Where FirstName = 'Leonard'
order by LastName;
```

Abbildung 50: Union-Operator über zwei Tabellen in zwei unterschiedliche Datenbanken

Man kann auch die zusammengesetzte View als eine feste View in der Datenbank hinterlegen und mittels Annotation über Hibernate auf diese zugreifen.

Sollte eine Analyse der Datenbankaufrufe erkennen, welche Daten vermehrt gebraucht werden, ist es möglich, die durch *Union*-Operator zusammengefassten Tabellen als eine materialisierte Sicht abzulegen und damit die Performanz der Anfrage signifikant zu erhöhen.

Durch das Verbinden der verschiedenen Tabellen ist nicht mehr ersichtlich, aus welcher Datenbank die Daten stammen. Aus diesem Grund ist es ratsam, eine weitere temporäre Spalte hinzuzufügen, welche die Herkunft der Daten anzeigt, denn mithilfe der ID und der Hilfsspalte mit Informationen zur Herkunft der Daten ist bekannt, wo der gewünschte Datensatz liegt. Somit kann beispielsweise gezielt ein *Update*- oder *Delete*-Statement an die Zieldatenbank geschickt werden, um diesen Datensatz zu aktualisieren. Abbildung 51 zeigt am Beispiel eines Code-Ausschnitts, wie der Verlauf eines *Update*-Befehls aussehen könnte.

```
01 Drop procedure updateProcedure;
02 CREATE PROCEDURE updateProcedure @RowNumber int,
                                @LastName varchar(30),
                                @FirstName nvarchar(30)
03
04 AS
05 DECLARE
06     @tempDatabase as varchar,
07     @tempPersonID int;
08     set @tempDatabase = 'initString';
09     set @tempPersonID = 0;
10
11 WITH GenericTable As
12 (
13     Select ROW_NUMBER() OVER (ORDER BY PersonID) As RowNumber,
14     tmpTbl.PersonID, tmpTbl.LastName, tmpTbl.FirstName, Datenbank
15     from
16         (
17             (SELECT *, '1' As 'Datenbank' FROM
18             DB1.dbo.persons)
19             UNION
20             (SELECT *, '2' As 'Datenbank' FROM
21             DB2.dbo.persons)
22         ) tmpTbl
23     select * from GenericTable
24     select @tempDatabase = Datenbank from GenericTable where
25     RowNumber = @RowNumber
26     select @tempPersonID = PersonID from GenericTable where
27     RowNumber = @RowNumber;
28
29 IF (@tempDatabase='1')
30 Begin
31     update DB1.dbo.persons
32     Set LastName = @LastName, FirstName = @FirstName
33     where PersonID = @tempPersonID;
34 END
35 ELSE
36 BEGIN
37     update DB2.dbo.persons
38     Set LastName = @LastName, FirstName = @FirstName
39     where PersonID = @tempPersonID;
40 END
41 GO
42
43 EXEC updateProcedure @RowNumber=4, @LastName='HermanNew',
44 @FirstName = 'PatrickNew';
```

Abbildung 51: Update, Delete-Prozedur

In Abbildung 51 wird in Zeile 01 eine Prozedur gelöscht für den Fall, dass eine Prozedur mit diesem Namen bereits existiert. In den Zeilen 05 bis 09 werden die Variablen deklariert. Von Zeile 11 bis 21 wird eine generische Tabelle erzeugt, die aus den Daten aus DB1 und DB2 mit dem *Union*-Operator in Zeile 17 bis 19 zusammengesetzt wird. In Zeile 17 und 19 wird mit dem String „Datenbank“ eine Spalte ergänzt und der Wert 1 bzw. 2 eingefügt, und zwar abhängig davon, aus welcher Datenbank die Daten bezogen worden sind. In Zeile 26 bis 38 wird

mithilfe einer *if*-Anweisung die richtige Datenbank, auf der sich der Datensatz befindet, lokalisiert. Anschließend wird der Datensatz durch den *Update*-Operator und mit den Argumenten, die der Prozedur mitgegeben worden sind, aktualisiert.

Wenn kein Kriterium existieren darf, das die anzufordernden Daten betitelt (aus welcher DB sollen die Daten genommen werden), dann wird jeder *Insert*-Operator immer an die Produktivdatenbank gehen und somit das direkte Einfügen neuer Daten in das Archiv durch den Anwender unterbinden. Bei jedem *Update* bzw. *Delete* muss ein Trigger⁹ geworfen werden, der die Logik beinhaltet, die Daten aus den verschiedenen Tabellen mit dem *Union*-Parameter zu verbinden, eine Hilfsspalte hinzufügen und anhand der Hilfstabelle und der ID die richtigen Datensätze zu manipulieren. Select-Anfragen müssten ohne Kriterium immer dafür sorgen, dass eine View aus den Tabellen aus beiden Datenbanken zusammengesetzt werden kann und daraus die Ergebnisse gebildet werden.

Durch die Nutzung von *Union*-Operator und mehreren *Join*-Operatoren wird die Performanz etwas langsamer.

⁹ Trigger sind Mechanismen, die aktiviert werden, wenn bestimmte Bedingungen eintreffen

6 Proof-of-Concept

Im Kapitel Proof-of-Concept geht es darum, die ausgearbeiteten Grundlagenkonzepte prototypisch zu entwickeln, um somit die Theorie in Form eines sogenannten Proof-of-Concept (Beweis) zu validieren. Dieses Kapitel gliedert sich in das Kapitel Entwicklungsumgebung, in der die verwendete Software beschrieben wird. Das zweite Kapitel bildet das Datenbankmodell, in dem ausgearbeitet wird, was für ein Datenmodell genutzt wurde und ebenso, wie dieses erstellt und mit Inhalt befüllt wurde. Das dritte Kapitel beschreibt die Umsetzung der in Kapitel 2.3.1 vorgestellten Grundlagen der Partitionierung in das entworfene Datenmodell.

6.1 Entwicklungsumgebung

Die prototypische Implementierung des Archivierungssystems untergliedert sich in zwei Bereiche: die Datenbank und die Objektorientierung. Die Lösung im objektorientierten Bereich wurde mit der integrierten Entwicklungsumgebung Eclipse Photon Candidate 3 (4.8.0RC3) umgesetzt. Die Java Enterprise Edition IDE wird speziell für Webentwicklungen¹⁰ genutzt. Als Applikationsserver wurde die JBoss-Enterprise-Application-Plattform mit der Version 7.0 von Red Hat¹¹ verwendet, die Hibernate von Haus aus mitgebracht hat.

Auf der Datenbankseite wurde der SQL Server 2017¹², die Entwickler-Version als relationales Datenbankmanagementsystem von Microsoft verwendet. Zur Verwaltung der Microsoft-SQL-Datenbank wurde der SQL Server Management Studio (SSMS) in der Version 15.0 18142.0¹³ benutzt. Zudem wurde der SQL Data Generator von Red Gate¹⁴ verwendet, um die Daten der Tabellen mit realistisch wirkenden Testdaten zu generieren.

Mit der Webanwendung Jira von Atlassian¹⁵ wurde ein Kanbanboard abgebildet. Primäres Ziel dieser Abbildung war es, den Durchlauf der Anforderungen im agilen Prozess zu erhöhen und die Fehler im Prozess zu minimieren.

6.2 Erstellung des Datenmodells

Die Erstellung des Datenmodells erfolgt in der Java-Applikation mithilfe von Hibernate. Klassen, die als Entitäten gekennzeichnet sind, werden mit Hibernate in eine relationale Datenbank gespeichert. Ist diese Entität nicht in der Datenbank zu finden, wird eine neue Tabelle angelegt. Zusätzlich werden bestimmte Annotationen verwendet, um beispielsweise Relationen festzulegen.

Das angelegte Datenmodell besteht aus vier Tabellen nämlich *Product*, *Customer*, *Orders* und *OrderItem*. Die Tabelle *Product* enthält die Spalte *PRODUCT_ID*, *ProductName* und *UnitPrice*. Die Tabelle *Customer* beinhaltet Kundendaten mit den Attributen *CUSTOMER_ID*,

¹⁰ <https://www.eclipse.org/>.)

¹¹ <https://developers.redhat.com/products/eap/overview>

¹² <https://www.microsoft.com/de-de/sql-server/sql-server-2017>

¹³ <https://docs.microsoft.com/de-de/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-2017>

¹⁴ <https://www.red-gate.com/products/sql-development/sql-data-generator/>

¹⁵ <https://www.atlassian.com/de/software/jira>

FirstName, *LastName*, *City*, *Country* und *Phone*. Die Tabelle *OrderItem* stellt die Bestellpositionen einer Bestellung dar. Diese enthält die Spalten *ORDER_ID*, *PRODUCT_ID*, die Spalte *Quantity* (gibt Anzahl an Käufen an) und die Spalte *UnitPrice* (Produktpreis). Die Tabelle *Orders* enthält alle Bestellungen, die mit den Spalten *ORDER_ID*, *OrderDate*, *CUSTOMER_ID* und *TotalAmount* getätigt worden sind. Der Gesamtpreis (*TotalAmount*) einer Bestellung errechnet sich aus der Summe der Bestellpositionen aus der Tabelle *OrderItem*, die das Produkt von *UnitPrice* und *Amount* ist. Die ID-Attribute in den jeweiligen Tabellen bilden den primären Schlüssel mit der Ausnahme, dass der primäre Schlüssel der *OrderItem*-Tabelle sich aus der Spalte *ORDER_ID* und der *PRODUKT_ID* zusammensetzt. Diese vier Tabellen repräsentieren eine Bestellung. Weil dieses Beispiel die Komplexität der Many-To-One- und Many-To-Many-Beziehungen aufzeigt, ist es als Vorzeigemodell geeignet. Abbildung 52 zeigt das aus den verwendeten Klassen entstandene Klassendiagramm.

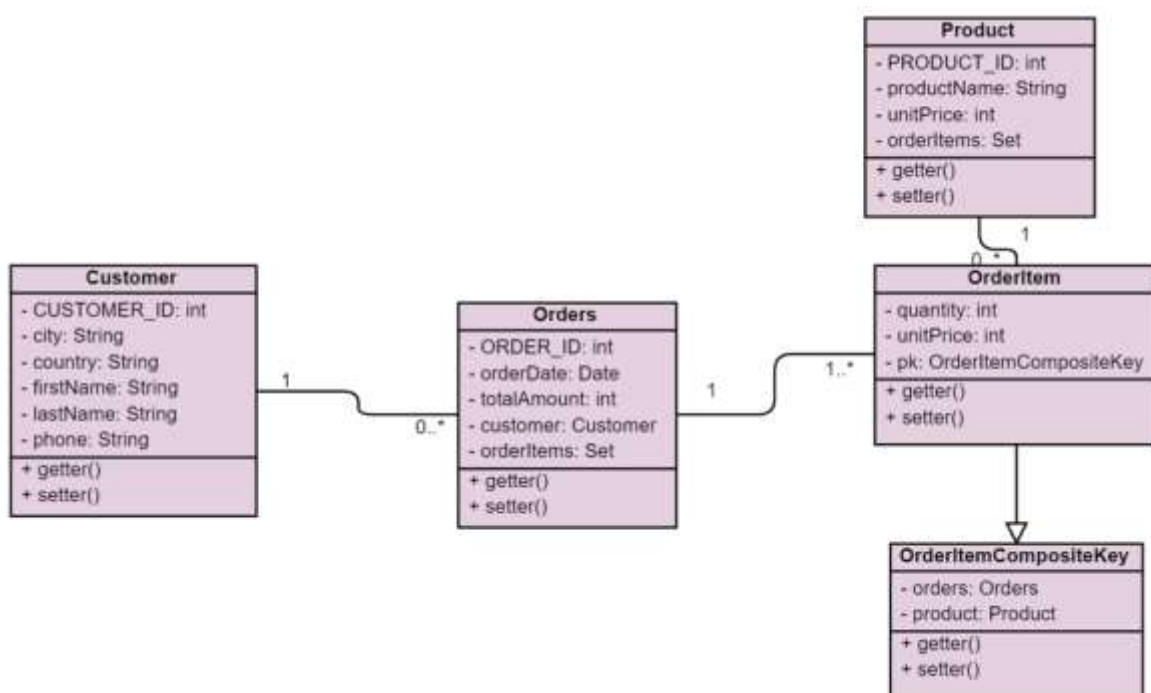


Abbildung 52: Klassendiagramm

Aus dem Klassendiagramm geht hervor, dass die Klasse *Orders* kein Attribut für einen Fremdschlüssel besitzt. Jedoch enthält die One-To-Many-Beziehung zwischen den Klassen *Customer* und *Orders* ein Attribut vom Typ *Customer*. Mit diesem Objekt wird in der Kind-Klasse (*Orders*) eine Referenz zu der Eltern-Klasse (*Customer*) hergestellt. Es gibt mehrere Arten, solch eine Relation herzustellen, doch ist dies nach Java-Champion¹⁶ Vlad Mihalcea, die effizienteste Methode (Mihalcea, Vlad Mihalcea, 2019).

Die Umsetzung eines zusammengesetzten Schlüssels in der *OrderItem*-Klasse wird mit einer Hilfstabelle realisiert. In der Klasse *OrderItem* wird eine Referenz auf die Klasse *OrderItemCompositeKey* erstellt. Diese Klasse enthält zwei Attribute, *orders* und *product*, welche die Referenz auf die Klasse *Orders* und die Klasse *Product* sind. Ebenfalls muss die Klasse *OrderItemCompositeKey* eine Java-Annotation *@Embedded* bekommen. Das Attribut vom Typ *OrderItemCompositeKey* in der Klasse *OrderItem* muss ebenso die Annotation *@Embedded* bekommen, denn damit wird Hibernate mitgeteilt, dass diese Klasse einen zusammengesetzten

¹⁶ <https://blogs.oracle.com/java/new-java-champions-in-2017>

Schlüssel verwendet (Mihalcea, Vlad Mihalcea, 2019). Zusätzlich wurde in dieser Klasse die Art der Beziehungen mit der Annotation `@ManyToOne` angegeben. Somit weiß Hibernate, sobald eine Referenz auf diese Klasse besteht, um welche Beziehung es sich handelt. Da alle Entitäten einer Session serialisiert werden müssen und diese Klasse als zusammengesetzter Schlüssel agiert, muss diese ebenfalls serialisiert werden.

Abbildung 53 zeigt einen Ausschnitt aus der Klasse *OrderItemCompositeKey.java*

```
@Embeddable
public class OrderItemCompositeKey implements
Serializable {
    private Orders orders;
    private Product product;

    @ManyToOne(cascade = CascadeType.ALL)
    public Orders getOrders() {
        return orders;
    }
    ...
    @ManyToOne(cascade = CascadeType.ALL)
    public Product getProduct() {
        return product; ...
    }
}
```

Abbildung 53: Ausschnitt aus der Klasse *OrderItemCompositeKey.java*

Eine Many-To-Many-Beziehung kann mittels der Annotation `@ManyToMany` realisiert werden. Hibernate erzeugt eine Hilfsklasse (*OrderItem*), die die Fremdschlüssel beider Tabellen (*Orders* und *Product*) besitzt. Da die Tabelle *OrderItem* in diesem Beispiel weitere Attribute, wie *UnitPrice* und *Quantity*, besitzt, ist die Realisierung dieser Verbindung nur möglich, indem die Many-To-Many-Beziehung in zwei Many-To-One-Beziehungen aufgeteilt wird (Mihalcea, vladmihalcea.com, 2019). Die unterteilten Many-To-One-Beziehungen gehen dabei einmal von *OrderItem* zu *Product* und einmal von *OrderItem* zu *Orders*.

Die Klasse *Orders* sowie auch die Klasse *Product* besitzen ein Attribut vom Typ *Set*, das Objekte vom Typ *OrderItem* aufnehmen kann. Dieses Set wird mit der Annotation `@OneToMany` versehen und enthält als Parameter *mappedBy* und *cascade*. *MappedBy* gibt Hibernate zu verstehen, dass diese Beziehung durch *primaryKey.orders* gehandhabt wird. In der Klasse *Product* ist der Wert, der *mappedBy* mitgegeben wird, *primaryKey.product*. Der Parameter *cascade* sorgt dafür, dass alle Kind-Elemente gespeichert bzw. aktualisiert werden, wenn die Eltern-Klasse (in dem Fall *Orders*) gespeichert wird. In Abbildung 54 ist ein Ausschnitt der Klasse *Orders.java* mit den Attributen und Annotationen zu sehen. Die Attribute und Annotationen realisieren die Beziehung zu der Klasse *Customer* und die Many-To-Many-Beziehung zwischen *Orders* und *Product*.

```

@Entity
@Table(name = "Orders")
public class Orders implements Serializable {    ...
    private Set<OrderItem> orderitems = new
HashSet<OrderItem>();
    private Customer customer;

    ...

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ORDER_ID")
    public int getOrderID() {
        return orderID;
    }

    @OneToMany(mappedBy = "primaryKey.orders", cascade =
CascadeType.ALL)
    public Set<OrderItem> getOrderitems() {
        return orderitems;
    }

    ...

    @ManyToOne
    @JoinColumn(name = "CUSTOMER_ID")
    public Customer getCustomer() {
        return customer;
    }

    ...

    @OneToMany(mappedBy = "primaryKey.orders", cascade =
CascadeType.ALL)

    public Set<OrderItem> getOrderitems() {
        return orderitems;
    } ...

```

Abbildung 54: Orders.java

Um die Tabellen mit Daten zu befüllen, wurde für die Tabellen *Product* und *Customer* der SQL Data Generator (SQL Data Generator, 2019) verwendet. Da diese keine Daten enthalten, die aus anderen Spalten oder Tabellen bezogen werden, konnte hierzu der Generator genutzt werden. Im GitHub-Repository (<https://github.com/tutzauel/MA>) können alle Informationen zur Erzeugung der Daten angesehen werden. Für die Tabellen *Orders* und *OrderItems* wurde ebenfalls der SQL Data Generator genutzt, um die Tabellen, die keine tabellenübergreifenden Daten besitzen, zu füllen. In der Tabelle *Orders* wurden die Werte für die Spalte *OrderDate* zufällig aus dem Jahr 2019 mithilfe dieser Software erzeugt. Auch wurde die Spalte *Quantity* in *OrderItem* ebenfalls mit der verwendeten Software generisch erzeugt.

Verwendet wird der SQL-Code aus Abbildung 55, der die Tabellen *Product* und *OrderItem* mit dem *Join*-Operator verbindet, um die Spalte *UnitPrice* aus der Tabelle *OrderItem* mit den richtigen Daten aus der Tabelle *Product* zu befüllen. Die Verbindung wird über die Spalten *OrderItem.Product_Id* und *Product.Product_Id* mittels des *Join*-Operators hergestellt. Der *Set*-Teil fordert, dass der *UnitPrice* aus *OrderItem* gleich dem *UnitPrice* aus Tabelle *Product* ist. Da die Tabellen über die genannten IDs verbunden werden und die Bedingungen des *Set*-Teils erfüllt werden, findet keine Verletzung der Fremdschlüsselbeziehung statt.

```
UPDATE dbo.OrderItem
SET dbo.OrderItem.UnitPrice =
dbo.Product.UnitPrice
FROM OrderItem
INNER JOIN dbo.Product
ON OrderItem.Product_Id =
dbo.Product.Product_Id
```

Abbildung 55: Inhalt der Spalte UnitPrice der Tabelle OrderItem aktualisieren

Damit die Spalte *TotalAmount* aus der Tabelle *Order* mit Werten, die aus der Summe der einzelnen Bestellpositionen aus *OrderItem* bestehen, befüllt werden kann, wurde der SQL-Code aus Abbildung 56 verwendet. Dieser Code verbindet beide Tabellen mithilfe der *ORDER_ID*, berechnet das Produkt von *UnitPrice* und *Quantity* und summiert mit der Aggregatsfunktion *Sum()* das berechnete Produkt für alle Positionen, in denen die *ORDER_ID* der Tabelle *OrderItems* mit der *ORDER_ID* der Tabelle *Orders* identisch ist.

```
UPDATE dbo.Orders
SET TotalAmount =
(SELECT SUM(a.UnitPrice * a.Quantity)
FROM dbo.OrderItem AS a
WHERE a.order_id = dbo.Orders.Order_Id
GROUP BY a.order_id)
```

Abbildung 56: Inhalt der Spalte TotalAmount der Tabelle Orders aktualisieren

Es wurden 10 000 Kunden angelegt, 100 000 Produkte und 300 000 Bestellungen mit insgesamt 1 Mio. Bestellpositionen. Der Zeitraum der Bestellungen (*OrderDate*) beschränkt sich auf das Jahr 2019. Die Tabelle *Quantity* in der Tabelle *OrderItem* wurde mit zufälligen Werten zwischen 1 und 10 versehen und der Preis der Produkte (*UnitPrice*) liegt zwischen 1 und 500. Das mit dem Microsoft SQL Server Management Studio erzeugte Datenbankdiagramm ist in Abbildung 57 mit den Primär- und Fremdschlüsselbeziehungen dargestellt.

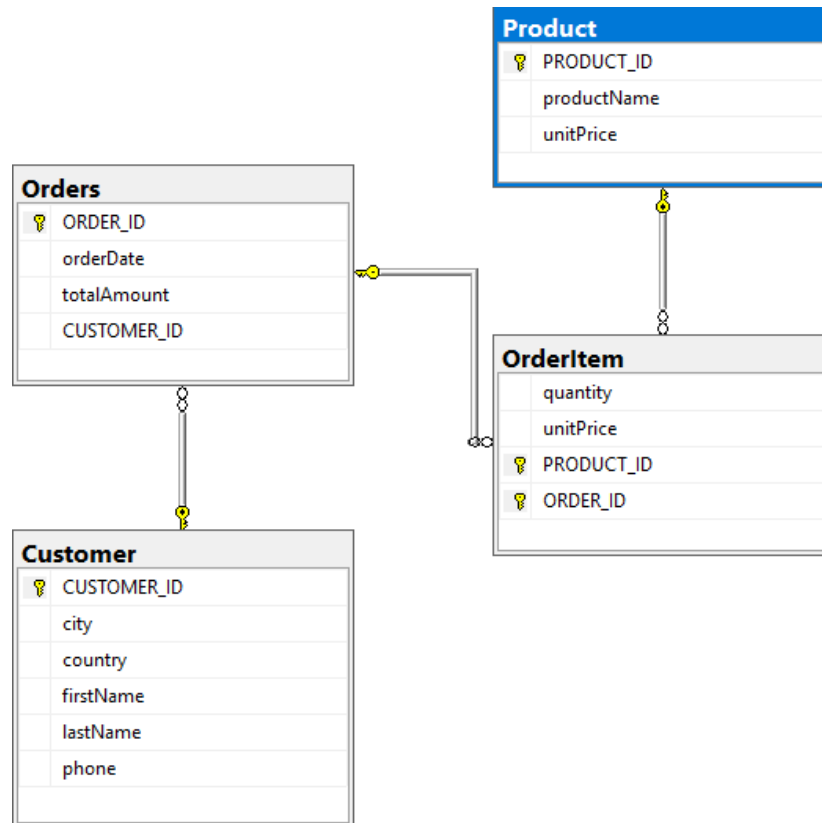


Abbildung 57: Datenmodell

6.3 Umsetzung der Partitionierung

Da für jeden Monat eines Jahres eine Partition erstellt werden soll, gibt es genau elf Grenzen, beginnend bei Februar ('20190201') bis Dezember ('20191201'). Datensätze für den Januar werden automatisch in die Partition 1 abgelegt, da diese kleiner ist als das erste Kriterium. Als Partitionsspalte wird die Spalte *OrderDate* der Tabelle *Orders* dienen. Zusätzlich wurde Range RIGHT verwendet, um in jeder Partition tatsächlich die Daten eines Monats zu speichern. Das Partitionsschema (Abbildung 58) veranlasst die Speicherung aller Datensätze einer Partition in die Dateigruppe [Partition].

```

-- Partitionsfunktion erstellen
Create Partition Function myPartitionFunction(datetime)
AS
Range RIGHT FOR VALUES
('20190201', '20190301', '20190401',
 '20190501', '20190601', '20190701', '20190801',
 '20190901', '20191001', '20191101', '20191201')

-- Partitionsschema erstellen
Create Partition Scheme myPartitionScheme AS
Partition myPartitionFunction ALL TO ([PRIMARY])
  
```

Abbildung 58: Partitionsfunktion und Partitionsschema

Die *OrderDate*-Spalte wird zur Partitionierung genutzt. Da ein Primary-Key auf der *ORDER_ID*-Spalte gesetzt ist, wurde von SQL Server automatisch ein gruppierter Index erzeugt. Für die Partitionierung ist es jedoch notwendig, dass der *Clustered-Index* auf die Spalte der Partitionsspalte gesetzt ist. Aus diesem Grund muss der primäre Schlüssel gelöscht und wieder neu erstellt werden. Weil von der Tabelle *OrderItem* eine Fremdschlüsselbeziehung zu dieser Spalte besteht, muss diese Fremdschlüsselbeziehung im ersten Schritt gelöscht werden. Durch die Fremdschlüsselbeziehung von *OrderItem* zu der Tabelle *Orders* ist es nicht möglich, einen zusammengesetzten primären Schlüssel aus der *ORDER_ID* und der Partitionsspalte *OrderDate* zu erzeugen. Dies würde voraussetzen, dass die Tabelle *OrderItem* die gleichen Spalten hat, aus denen der primäre Schlüssel besteht, und dieser Zustand ist nicht gegeben.

Aus diesem Grund wird ein Non-Clustered Primary-Key erzeugt, der das Ziel der Fremdschlüsselbeziehung aus der Tabelle *OrderItem* sein wird. Weil dieser keinen Index erzeugt, ist es möglich, einen Clustered-Index auf die Spalte der Partitionsspalte zu setzen und damit Partitionen zu schaffen. In Abbildung 59 ist ein Code-Ausschnitt zu sehen, der diesen Prozess beschreibt.

```
-- FK löschen
alter table orderItem drop constraint FK_ORDERITEM_REFERENCE_ORDER
GO

-- PK löschen
alter table orders drop constraint PK_ORDERS
GO

-- Nicht gruppierter PK erstellen
alter table orders add constraint PK_ORDERS_NONCLUSTERED PRIMARY KEY
NONCLUSTERED (order_id)
ON [PRIMARY]
GO

-- Gruppierter Index erstellen
CREATE CLUSTERED INDEX INDEX_PARTITIONCOL_ORDERDATE ON orders
(OrderDate)
ON myPartitionScheme (orderDate)
GO

-- FK neu setzen
alter table orderitem add constraint FK_ORDERITEM_REFERENCE_ORDER
foreign key (order_id) references Orders (order_id)
GO
```

Abbildung 59: Die Schritte Löschen sowie Setzen der Schlüssel zur Partitionierung

Die Validierung, dass die Schritte korrekt durchgeführt wurden und sich wie erwartet verhalten, zeigt Abbildung 60. Sie veranschaulicht das Ergebnis und zeigt Informationen über die Partitionierung. *Objektname* beschreibt, um welche Tabelle es sich handelt. *Indexname* beschreibt den Index auf der genutzten Partitionierungsspalte und dass dieser ebenfalls partitioniert wurde. Im Vergleich sieht man in Zeile 13 den nicht gruppierten Primärschlüssel, der nicht partitioniert wurde. Darüber hinaus kann der Abbildung ebenfalls entnommen werden, dass dieser Schlüssel die kompletten 300 000 Datensätze hält, wobei der gruppierte Index unterteilt wurde und

jeweils ca. 25 000 Datensätze hält. Zusätzlich lässt sich aus der Abbildung erkennen, wie viele Partitionen es gibt und wie viel Datensätze jede enthält¹⁷.

Ergebnisse		Meldungen			
	objectname	indexname	partition_id	partition_number	rows
1	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043498496	1	25377
2	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043564032	2	23069
3	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043629568	3	25734
4	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043695104	4	24737
5	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043760640	5	25436
6	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043826176	6	24851
7	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043891712	7	25546
8	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594043957248	8	25680
9	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594044022784	9	24822
10	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594044088320	10	25702
11	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594044153856	11	24482
12	Orders	INDEX_PARTITIONCOL_ORDERDATE	72057594044219392	12	24564
13	Orders	PK_ORDERS_NONCLUSTERED	72057594044284928	1	300000

Abbildung 60: Validierung der Partitionierung

Nun wird in der Java-Applikation mittels Hibernate eine weitere Bestellung angelegt und in der Datenbank gespeichert. Die Partitionsfunktion wird diesen Datensatz überprüfen und das Datenbank-Management-System wird diesen Datensatz abhängig von dem Wert in der Spalte *OrderDate* in die entsprechende Partition setzen. Alle Klassen zum Erzeugen des Datenmodells, wie auch die SQL-Skripte, um die Partitionierung einzurichten befinden sich im GitHub-Repository (<https://github.com/tutzaue/MA>).

¹⁷ <https://www.mssqltips.com/sqlservertip/2888/how-to-partition-an-existing-sql-server-table/>

7 Test und Vergleich

In diesem Kapitel wird die Methode der Partitionierung, die im Kapitel Proof-of-Concept implementiert wurde, einem Performanztest unterzogen. Verglichen wird die Performanz einer partitionierten Datenbank mit der Performanz einer nicht partitionierten Datenbank. Dabei werden auf beiden zu untersuchenden Objekten dieselben Tests durchgeführt.

Alle Tests wurden auf demselben Rechner absolviert, um das Ergebnis möglichst authentisch zu halten. Ebenso befindet sich der SQL Server mit seinen Datenbanken lokal auf demselben Rechner. Es folgt eine Auflistung der Hardware vom Rechner in Tabelle 14, die für den Test genutzt wurde:

Hardware	Beschreibung
Laptop	Xtreme Performance System (XPS) 13 (9380)
Betriebssystem	Windows 10 Home-HE (64 Bit)
CPU	Intel(R) Core(TM) i7-8565U Prozessor der 8. Generation (8M Cache, bis zu 4,6 GHz, 4 Kerne)
Ram	16GB LPDDR3 2133MHz
Festplatte	512 SSD: 512GB PCIe M.2 Solid State Festplatte
Grafikkarte	Grafik: Intel(R) UHD Grafik 620

Tabelle 14: Technische Daten der Testumgebung

Die getesteten Abfragen waren realistisch und an die Realität angelehnt. Dieselben Abfragen wurden abwechselnd an die partitionierten und nicht partitionierten Datenbanken gestellt. Die Dauer einer jeden Abfrage wurde mittels Statistik-Metriken analysiert und der passenden Methode (partitioniert bzw. nicht partitioniert) zugeordnet. Die Ergebnisse der Untersuchung wurden einander gegenübergestellt. Zusätzlich wird der Ausführungsplan genau untersucht, weil dieser Informationen über die Zusammensetzung der Ergebnisse enthält. Ebenfalls beinhaltet der Ausführungsplan Auskünfte darüber, wie Joins abgearbeitet wurden, ob und wie gesetzte Indizes genutzt wurden und welche optimal sind.

Es wurden drei Datenbanken (*sample_db1*, *sample_db2*, *sample_db3*) erstellt, die nicht partitioniert sind, und drei weitere Datenbanken (*sample_db1_part*, *sample_db2_part*, *sample_db3_part*), die partitioniert sind. Alle Datenbanken enthalten die Tabellen *Customer*, *Product*, *Orders* und *OrderItem* und sind durch die gleichen Schlüssel miteinander verbunden. Die Datenbanken *sample_db1* und *sample_db1_part* (und folglich auch die Datenbanken *sample_db2* VS. *sample_db2_part* und *sample_db3* VS. *sample_db3_part*) treten als Kontrahenten im Test gegeneinander an und haben beide die gleiche Anzahl an Datensätzen in den jeweiligen Tabellen. Die Anzahl der Datensätze in den Datenbanken steigt mit der Nummer der Datenbank, so sind in beiden DB2 10-mal so viele Datensätze wie in beiden DB1 und folglich auch 10-mal mehr Datensätze in beiden DB3 als in DB2.

In der Tabelle 15 ist eine genaue Auflistung der Anzahl der Datensätze in den Tabellen in den jeweiligen Datenbanken zu sehen. Es gilt zu beachten, dass in diesem Test gleichwertige und nicht identische Daten benutzt wurden. Beispielsweise kann die Tabelle *Customer* die erste ID in der einen Datenbank dem *Vornamen* Mia zuweisen und in der gleichen Tabelle einer anderen Datenbank den *Vornamen* Noah.

	Sample_db1/_part	Sample_db2/_part	Sample_db3/_part
Customer	10.000	100.000	1.000.000
Products	100.000	1.000.000	10.000.000
Orders	300.000	3.000.000	30.000.000
OrderItems	1.000.000	10.000.000	100.000.000
Größe der DB ~	0,5 GB	4 GB	40 GB

Tabelle 15: Anzahl der Datensätze in den Datenbanken und deren Größe in Gigabyte

Wie in Kapitel 2.3.5 wurden auch diese Datenbanken nach dem gleichen Schema mit Daten befüllt. In der Tabelle *Orders* werden nur Daten aus dem Jahr 2019 enthalten sein. Die Spalte *Quantity* aus der Tabelle *OrderItem* wird einen zufälligen Wert zwischen 1 und 10 einnehmen. Ebenso der *UnitPrice* der Tabelle *Product*, der einen zufälligen Wert zwischen 1 und 500 bekommt.

Die genauen Auszüge der Protokolle wie die Daten, welche Daten und Datentypen eingefügt wurden, finden sie im GitHub-Repository (<https://github.com/tutzaue1/MA>).

7.1 Performanztest

In diesem Kapitel geht es darum, konkrete Abfragen gegeneinander laufen zu lassen und die Zeit bis zur Rückgabe zu messen. Die hier verwendeten Queries wurden mit dem Softwareteam und dem Datenbank-Administrator der Firma Köhl erstellt und sind in Tabelle 16 zu sehen. Die Tabelle untergliedert sich in die Spalten *Nr.* und *Aufgabe*. Die Spalte *Nr.*, die die Nummer der Queries repräsentiert, wird im späteren Verlauf für eine vereinfachte Darstellung verwendet werden. Die Spalte *Aufgabe* gibt an, was die Abfrage leisten soll.

Query Nr.	Aufgabe
1	Ausgabe aller Datensätze der Tabelle <i>Orders</i>
2	Ausgabe aller Datensätze der Spalten <i>CUSTOMER_ID</i> , <i>lastName</i> , <i>orderDate</i> , <i>totalAmount</i> , <i>PRODUCT_ID</i> , <i>productName</i> und <i>unitPrice</i> aus den Tabellen <i>Customer</i> , <i>Orders</i> , <i>OrderItem</i> und <i>Products</i>
3	Ausgabe aller Datensätze der Spalten <i>CUSTOMER_ID</i> , <i>lastName</i> , <i>orderDate</i> , <i>totalAmount</i> , <i>PRODUCT_ID</i> , <i>productName</i> und <i>unitPrice</i> aus den Tabellen <i>Customer</i> , <i>Orders</i> , <i>OrderItem</i> und <i>Products</i> , wo <i>orderDate</i> den Wert '20190603 11:51:40:840' hat
4	Ausgabe aller Datensätze der Spalten <i>CUSTOMER_ID</i> , <i>lastName</i> , <i>orderDate</i> , <i>totalAmount</i> , <i>PRODUCT_ID</i> , <i>productName</i> und <i>unitPrice</i> aus den Tabellen <i>Customer</i> , <i>Orders</i> , <i>OrderItem</i> und <i>Products</i> , wo <i>orderDate</i> den Wert größer/ gleich '20190603 11:51:40:840' und kleiner/gleich '20190903 11:51:40:840' hat

5	Ausgabe aller Datensätze der Spalten <i>CUSTOMER_ID</i> , <i>lastName</i> , <i>orderDate</i> , <i>totalAmount</i> , wo <i>orderDate</i> den Wert größer/gleich '20190603 11:51:40:840' und kleiner/gleich '20190903 11:51:40:840' hat
---	---

Tabelle 16: Nummer und Aufgabe der entsprechenden Anfrage

In Tabelle 17 werden die in Tabelle 16 beschriebenen Aufgaben in SQL-Befehle umgewandelt, sodass die Anfrage von einer Datenbank interpretiert werden kann. Tabelle 17 hat zwei Spalten, eine mit der Query-Nummer und eine Spalte *Abfrage*, die den SQL-Befehl zur Ausführung beinhaltet.

Query Nr.	Abfrage
1	<code>SELECT * FROM dbo.Orders</code>
2	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUSTOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.ORDER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PRODUCT_ID</code>
3	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUSTOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.ORDER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PRODUCT_ID WHERE b.orderDate = '20190603 11:51:40:840'</code>
4	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUSTOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.ORDER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PRODUCT_ID WHERE b.orderDate >= '20190603 11:51:40:840' AND b.orderDate <= '20190903 11:51:40:840'</code>
5	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUSTOMER_ID WHERE b.orderDate >= '20190603 11:51:40:840' AND b.orderDate <= '20190903 11:51:40:840'</code>

Tabelle 17: Nummer und die Anfrage in der SQL-Sprache

Jede Anfrage an die Datenbank wurde 5-mal ohne Messung durchgeführt. Dabei wurde die Funktion *DBCC DROP CLEANBUFFERS* verwendet, die den Cache leert, um zwischengespeicherte Objekte zu entfernen. Damit wurde sichergestellt, dass die Anfragen tatsächlich ihre Daten aus der Datenbank beziehen und nicht aus dem Cache, was zu Verfälschung der

Rückgabezeit führen könnte. Die MSSQL-Funktion STATISTICS TIME wurde verwendet, um die Zeit in Millisekunden zu messen. Da die Rückgabe dieser Funktion von der aktuellen Belastung des Systems abhängig ist, wurde jede Anfrage 5-mal an die Datenbank gesendet, wobei jedes Mal die Dauer der Anfrage bis zur Rückgabe des Ergebnisses aufgezeichnet wurde. Diese Werte wurden miteinander addiert und durch die Anzahl der Durchläufe dividiert, um einen Durchschnittswert zu ermitteln.

In Tabelle 18 sind die Durchschnittszeiten der Abfragen an die partitionierten sowie die nicht partitionierten Datenbanken festgehalten. Ein Excel-Sheet mit dem vollständigen Testbericht befindet sich im Anhang. Außerdem wurde zum Zwecke der Übersichtlichkeit das Präfix *sample* in den Datenbanknamen entfernt.

Query Nr.	Dauer der Rückgabe einer Anfrage in Millisekunden (ms)					
	db1	db1_part	db2	db2_part	db3	db3_part
1	1598,6	1541,4	14133,4	14124,4	141460,6	140502
2	7275	7130,4	70946,4	70924,8	7181226,6	7103451,4
3	23,2	3,8	89,4	5,6	855,6	7,4
4	1899,6	1837,4	18420,6	17477,6	179823,8	160779,8
5	451,4	434	4221,4	4124,6	43158	41592,2

Tabelle 18: Zusammenfassung des Performanztests (Durchschnittswerte)

Eine Grafik in Abbildung 66 veranschaulicht das Ergebnis des Performanztests aus Tabelle 17, in der der Performanzunterschied am deutlichsten zu sehen war. In Abbildung 61 und 62 ist die Zeit in Millisekunden durch die Y-Achse dargestellt, während die X-Achse die Datenbanken angibt. Die Query Nummer 3 hatte bei der Anfrage ein genaues Zeitkriterium, wodurch die Partitionierung ihr volles Potenzial ausschöpfen konnte. Zusätzlich ist in Abbildung 61 zu erkennen, dass mit steigender Datenmenge die Performanz der Partitionierung nur minimal ansteigt.

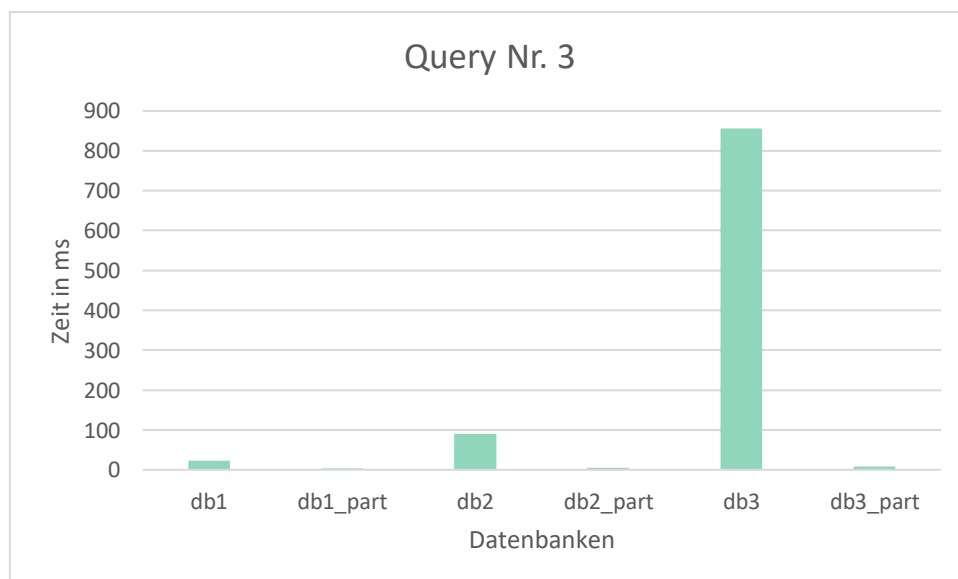


Abbildung 61: Ergebnis Query Nummer 3

Im Gegensatz dazu lässt Abbildung 62 erkennen, dass das Ergebnis der Query Nummer 3 unabhängig von der Datenmenge eine ähnliche Ausführungszeit aufweist. Dies ist darauf

zurückzuführen, dass in dieser Query kein Gebrauch von der Partitionsspalte *orderDate* aus der Tabelle *Orders* gemacht wurde. In diesem Fall kann von der Partition kein Nutzen gewonnen werden, was die gleiche Dauer für die Rückgabe der Anfrage erklärt.

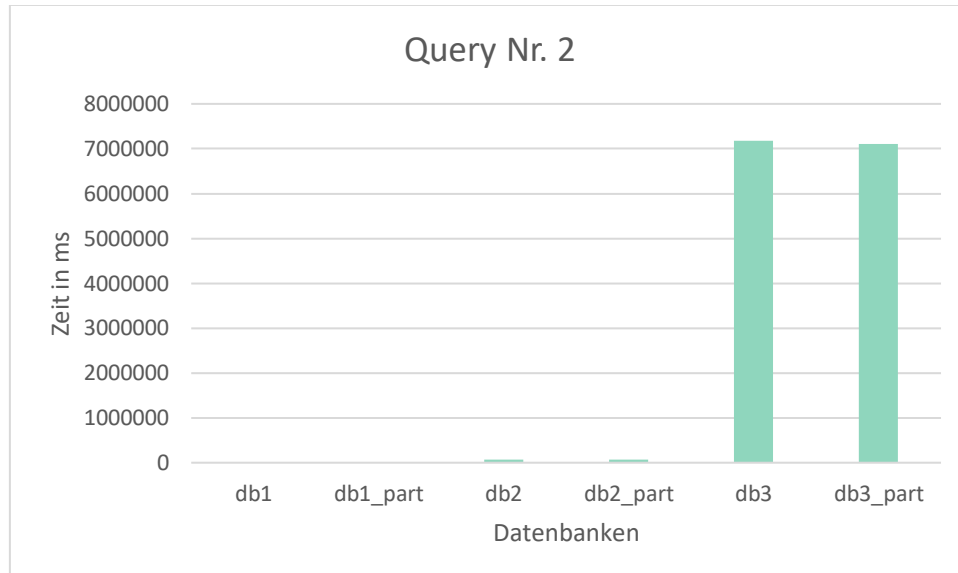


Abbildung 62: Ergebnis Query Nummer 2

7.2 SQL-Ausführungsplan

In diesem Teil des Vergleichs werden Queries im gleichen Batch gestartet. Dies hat den Vorteil, dass man bei erfolgreich ausgeführten Queries sehen kann, wie viel Prozent des Batches bei der jeweiligen Anfrage genutzt wurden. Damit kann man die Performance einer Abfrage in Abhängigkeit zu den Ressourcen, die vom Batch verwendet wurden, betrachten. Daraus wird ersichtlich, welchen prozentualen Wert an Ressourcen die Anfrage verbraucht hat. Zusätzlich wird an dieser Stelle der Ausführungsplan, der die genaue Abwicklung der Queries angibt, untersucht. Aus dem Ausführungsplan lässt sich entnehmen, wo welche Joins verwendet wurden, wie diese miteinander verbunden sind oder wie die gesetzten Queries genutzt wurden.

Zu jeglichen Schritten in diesem Prozess wird ein Prozentsatz angegeben, der veranschaulicht, wie viel Performanz im Verhältnis zu Batch verbraucht wurde.

In Abbildung 63 wurde die Query Nummer 3 an die Datenbank *sample_db1* und *sample_db1_part* geschickt. Die weiteren Ausführungspläne sind im GitHub-Repository einzusehen (<https://github.com/tutzaue/MA>). Die Ergebnisse der Ausgaben ändern sich nicht mit der Datenmenge, sondern bleiben proportionell zum Batch. Aus diesem Grund wird hier nur die *sample_db1* bzw. *sample_db1_part* betrachtet.

Diese zeigt, dass die Anfrage an die partitionierte Datenbank nur ein Prozent des Batches verwendet hat, wohingegen die Anfrage an die Datenbank ohne eine Partitionierung 99 % gebraucht hat. Es ist ebenfalls zu sehen, dass von SQL vorgeschlagen wird, einen nicht gruppierten Index aufzunehmen, um diese Abfrage zu verbessern.

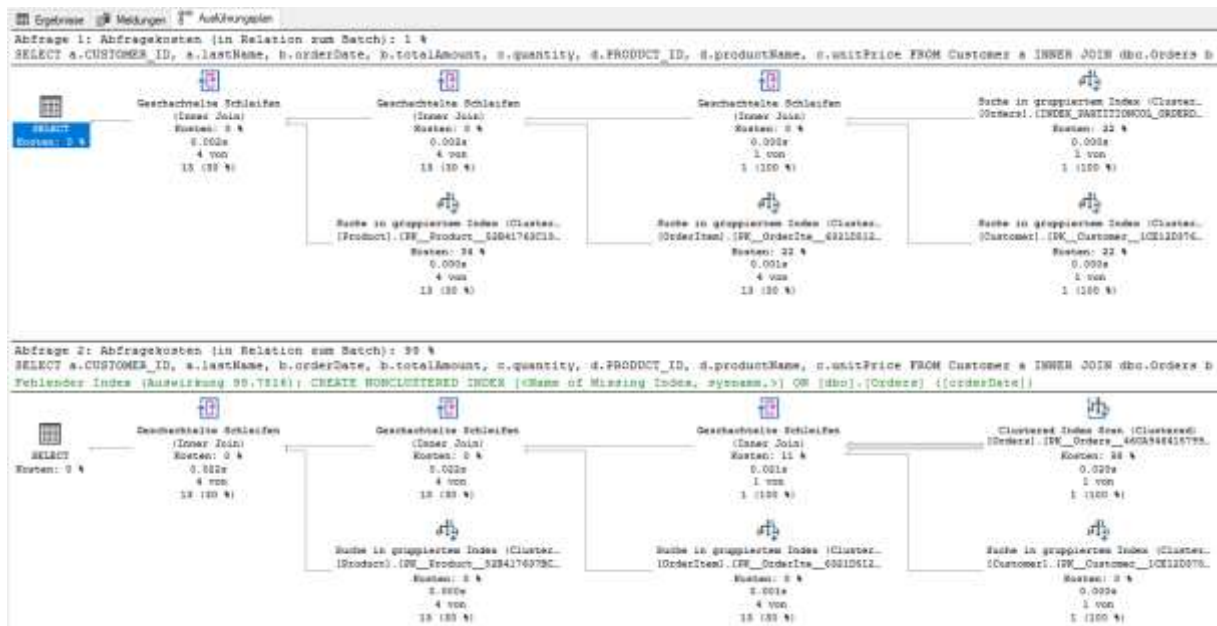


Abbildung 63: Ausführungsplan der Query Nummer 3

Abbildung 64 ist zu entnehmen, dass die partitionierte Variante 49 %, hingegen die nicht partitionierte Variante 51 % verbraucht hat. Informationen zu jedem Knotenpunkt lassen sich ebenfalls aufrufen. Somit kann man im Knotenpunkt der Tabelle *Orders* Informationen zu diesem erhalten. Hier sieht man, dass diese Tabelle partitioniert wurde (partitioniert: True) und in dieser Abfrage vier Partitions durchlaufen wurden (tatsächliche Partitionsanzahl: 4).

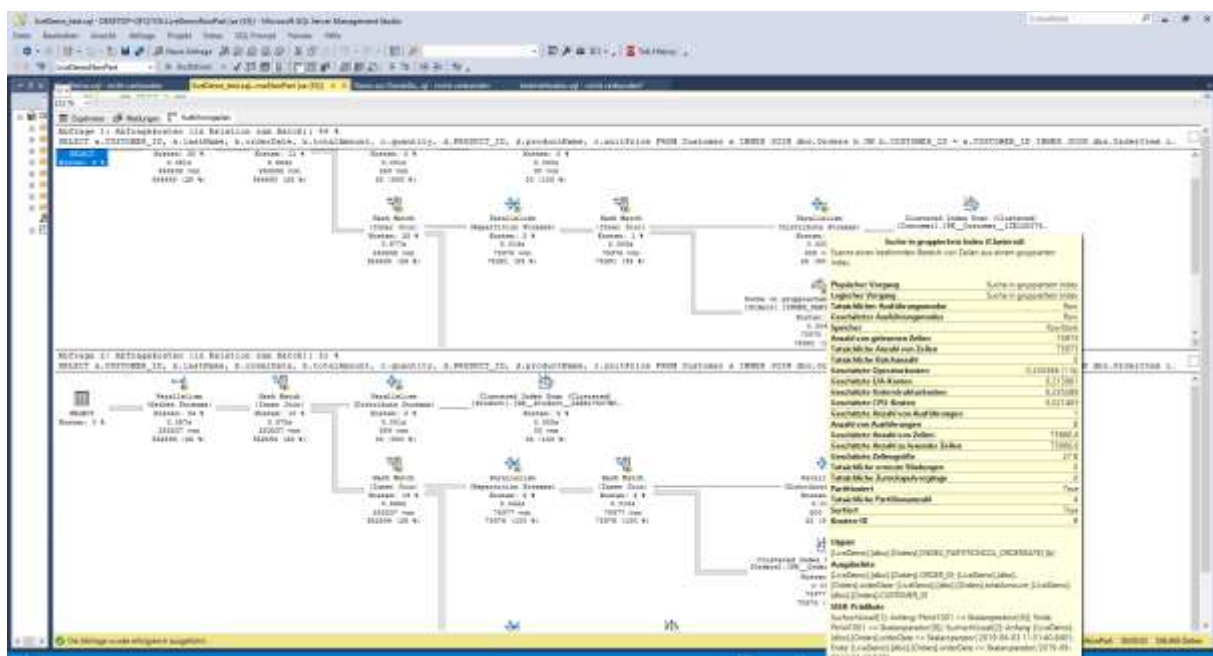


Abbildung 64: Ausführungsplan der Query Nummer 4

8 Zusammenfassung und Ausblick

Diese Arbeit entstand in Kooperation mit der Firma Köhl, die den Zugang zu ihren Daten optimiert wissen wollte. Die Behebung des Problems sollte jedoch unter bestimmten Bedingungen erfolgen. Zunächst wurde vermutet, dass die umfangreiche Menge an Daten die Produktivität des Systems herabsetzt. Die Recherche jedoch hat gezeigt, dass nicht die Datenmenge allein der Grund für die entschleunigte Performanz war, sondern auch andere Faktoren. Darauf basierend wurden Konzepte ausgearbeitet, die ein bestehendes System in seiner Performanz verbessern sollten. Nach ausgiebiger Untersuchung des gegebenen Sachverhalts hat sich die Methode der Partitionierung als geeignet erwiesen und aus diesem Grund wurde sie prototypisch implementiert.

Im Zuge des wissenschaftlichen Vorgehens wurden zuerst die Grundlagen in den Bereichen Datenbanken, objektrelationale Abbildung und Partitionierung dargeboten. Darauf aufbauend wurde der Aspekt des Designs, das Themen wie Normalisierung bzw. Denormalisierung und die richtige Nutzung von Indizes beinhaltet, erläutert. Ebenfalls wurde das Softwaretuning, das sich mit der Verbesserung von Hibernate und dem SQL Server befasst, behandelt. Die Möglichkeit der Performanzsteigerung durch Anpassung der beiden Aspekte wurde in dem Zusammenhang untersucht und bestätigt. Anschließend wurde der Fokus auf verschiedene Archivierungsverfahren und deren Grundzüge gelegt. Nachdem alle gesammelten Fakten abgewogen wurden, wurde die Methode der Partitionierung als passend befunden. Partitionierung lässt sich in ein bereits bestehendes System integrieren, ohne dieses verändern zu müssen, da die Logik sich komplett auf der Datenbank befindet. Zusätzlich können die Inhalte der Partitionen durch Neuausrichtung der Zeiger (Pointer) augenblicklich verschoben werden, was Sperrzeiten innerhalb einer Tabelle auf ein Minimum reduzieren kann. Demnach wurde die Methode der Partitionierung prototypisch implementiert. Schließlich wurden Tests durchgeführt, die die Performanzsteigerung des Systems beweisen sollten, wobei partitionierte Datenbanken gegen nicht partitionierte Datenbanken getestet wurden. In keinem der Testfälle waren die partitionierten Datenbanken langsamer als die nicht partitionierten, sondern mindestens gleich schnell oder schneller. Als besonders effektiv erwies sich die Partitionierung in Anfragen mit einer Zeitangabe als Kriterium, in diesen Fällen waren partitionierte Datenbanken um ein Vielfaches schneller als nicht partitionierte. Daraus wurde entnommen, dass die Partitionierung eine Lösung für die Firma Köhl bietet und gleichzeitig alle vorgegebenen Voraussetzungen erfüllt.

Der Mehrwert dieser Arbeit liegt in der Zugänglichkeit der Partitionierung. Da sie sich nachträglich in bestehende Systeme einbauen lässt, kann jedes System um die Partitionierung erweitert werden. Ab SQL Server 2016 ist sie darüber hinaus frei zugänglich, was sie umso attraktiver macht. Es ist eine Methode, von deren Benutzung viele Unternehmen profitieren könnten.

Als Zukunftsvision wäre es wünschenswert, die Partitionierung in dem System der Firma Köhl integriert zu wissen.

Literatur

- Amdtown. (04. 2019). Abgerufen am 13. 08. 2019 von <https://www.amdtown.com/WPD6MYN2/>
- Amdtown. (08. 05. 2019). Abgerufen am 13. 08. 2019 von <https://www.amdtown.com/2PG537KP/>
- Augsten, S. (23. 03. 2018). *Dev-Insider*. Abgerufen am 13. 08. 2019 von <https://www.dev-insider.de/was-ist-xml-a-692619/>
- Barry, D. K. (2010 - 2019). *Service-Architecture*. Abgerufen am 13. 08. 2019 von https://www.service-architecture.com/articles/object-oriented-databases/odbms_faq.htm
- Barry, D. K. (2019). *Service Architecture*. Abgerufen am 13. 08. 2019 von https://www.service-architecture.com/articles/database/concurrency_control_and_locking.html
- Bauer, C., & King, G. (2007). *Java Persistence mit Hibernate*. Carl Hanser Verlag .
- Bauer, R. (02. 08. 2018). *Blackblaze*. Abgerufen am 13. 08 2019 von <https://www.backblaze.com/blog/data-backup-vs-archive/>
- Begerow, M. (2019). *Datenbanken-Verstehe.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/>
- Begerow, M. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/erste-normalform/>
- Begerow, M. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/zweite-normalform/>
- Begerow, M. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/dritte-normalform/>
- Begerow, M. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/vierte-normalform/>
- Begerow, M. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/datenmodellierung/normalisierung/fuenfte-normalform/>
- Brumm, B. (26. 07. 2018). *Databasestar*. Abgerufen am 13. 08. 2019 von <https://www.databasestar.com/sql-views/>
- Cheriton, D. R. (1985). Preliminary Thoughts on Problem-oriented Shared Memory: A Decentralized Approach to Distributed Systems. (S. University, Hrsg.) *Operating Systems Review*, 19(4), S. 26-33.
- Dartmouth.edu. (02. 2004). Abgerufen am 13. 08. 2019 von <https://www.dartmouth.edu/~bknauff/dwebd/2004-02/DB-intro.pdf>

- Database.Guide.* (22. 06. 2016). Abgerufen am 13. 08. 2019 von <https://database.guide/what-is-a-document-store-database/>
- DB-Engines.* (08. 2019). Abgerufen am 13. 09. 2019 von <https://db-engines.com/de/ranking>
- Doward, M. (14. 03. 2019). *ptr.co.uk*. Abgerufen am 13. 08. 2019 von <https://www.ptr.co.uk/blog/sql-server-aggregate-functions-over-clause>
- Gebhardt, K. F. (05. 12. 2018). Datenbanken. Stuttgart, Baden-Württemberg, DE. Abgerufen am 13. 08. 2019 von <http://www.lehre.dhbw-stuttgart.de/~kfg/db/db.pdf>
- Goram, M. (12. 02. 2017). *datenbanken-verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/lexikon/2-phases-sperrprotokoll/>
- Gupta, L. (01. 07. 2013). *Howtodoinjava*. Abgerufen am 13. 08. 2019 von <https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/>
- Haß, K. (2019). *Datenbanken-Verstehen.de*. Abgerufen am 13. 08. 2019 von <http://www.datenbanken-verstehen.de/lexikon/hibernate-query-language/>
- Hibernate Community Documentation.* (n.d.). Abgerufen am 13. 08. 2019 von <https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch03.html>
- Hibernate Community Documentation.* (n.d.). Abgerufen am 08.13 2019 von <https://docs.jboss.org/hibernate/orm/5.0/manual/en-US/html/ch01.html#tutorial-associations>
- Hibernate Community Documentation.* (n.d.). Abgerufen am 13. 08. 2019 von <https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#ftn.d0e2213>
- Horn, T. (2007). *torsten-horn.de*. Abgerufen am 19. 08. 2019 von <https://www.torsten-horn.de/techdocs/sql-vererbung.htm>
- Horn, T. (2008). *Torsten-Horn.de*. Abgerufen am 13. 08. 2019 von <https://www.torsten-horn.de/techdocs/java-annotations.htm>
- Horn, T. (2010). *Torsten-Horn.de*. Abgerufen am 13. 08. 2019 von <https://www.torsten-horn.de/techdocs/java-jpa.htm>
- IT-Visions.de.* (16. 07 2017). Abgerufen am 13. 08. 2019 von https://www.it-visions.de/glossar/alle/838/Was_ist_ObjektRelationales_Mapping_ORM.aspx
- ITWissen.Info.* (20. 12 2016). Abgerufen am 13. 08. 2019 von <https://www.itwissen.info/Persistenz-persistence.html>
- Jansen, T. (2019). *Thoughts On Java*. Abgerufen am 13. 08. 2019 von <https://thoughts-on-java.org/how-to-join-unrelated-entities/>
- Jansen, T. (2019). *Thoughts On Java*. Abgerufen am 13. 08. 2019 von <https://thoughts-on-java.org/fetch-multiple-entities-id-hibernate/>
- JPA Performance Benchmark (JPAB).* (2012). Abgerufen am 13. 08. 2019 von http://www.jpab.org/Running_and_Results.html
- Juneau, J. (2018). The Query API and JPQL. In: Java EE 8 Recipes. Berkeley, CA: Apress.
- Klößner, K. (23. 03. 2016). *Im Vergleich: NoSQL vs. relationale Datenbanken*. Abgerufen am 13. 08 2019 von http://pi.informatik.uni-siegen.de/Mitarbeiter/mrindt/Lehre/Seminare/NoSQL/einreichungen/NOSQLTEC-2015_paper_9.pdf
- Lahres, B., & Rayman, G. (2006). *Openbook.Rheinwerk-Verlag.de*. Abgerufen am 13. 08. 2019 von http://openbook.rheinwerk-verlag.de/oo/oo_04_strukturvonooprogrammen_02_003.htm
- Lipinski, K. (08. 11. 2013). *ITWissen.info*. Abgerufen am 13. 08. 2019 von <https://www.itwissen.info/Hibernate-hibernate.html>
- Lipinski, K. (06. 08. 2016). *ITWissen.info*. Abgerufen am 13. 08. 2019 von <https://www.itwissen.info/Verteilte-Datenbank-distributed-database-DDB.html>

- Luber, S. (17. 09. 2017). *BigData Insider*. Abgerufen am 13. 08. 2019 von <https://www.bigdata-insider.de/was-ist-eine-relationale-datenbank-a-643028/>
- Luber, S. (15. 11. 2018). *BigData Insider*. Abgerufen am 13. 08. 2019 von <https://www.bigdata-insider.de/was-ist-acid-a-776182/>
- Mehra, A. (06. 09. 2017). *Akhil Mehra*. Abgerufen am 13. 08. 2019 von <https://dzone.com/articles/understanding-the-cap-theorem>
- Microsoft SQL Docs*. (03. 02. 2017). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/configure-the-max-degree-of-parallelism-server-configuration-option?view=sql-server-2017>
- Microsoft SQL Docs*. (01. 09. 2017). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/en-us/sql/relational-databases/performance/database-engine-tuning-advisor?view=sql-server-2017&viewFallbackFrom=sql-server-2017>
- Microsoft SQL Docs*. (16. 08. 2017). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/en-us/sql/relational-databases/automatic-tuning/automatic-tuning?view=sql-server-2017>
- Microsoft SQL Dokumentation*. (20. 01 2016). Von <https://docs.microsoft.com/de-de/sql/relational-databases/partitions/partitioned-tables-and-indexes?view=sql-server-2017> abgerufen
- Microsoft SQL Dokumentation*. (14. 03. 2017). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/de-de/sql/relational-databases/stored-procedures/stored-procedures-database-engine?view=sql-server-2017>
- Microsoft SQL Dokumentation*. (19. 11. 2018). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/de-de/sql/relational-databases/views/create-indexed-views?view=sql-server-2017>
- Microsoft SQL Dokumentation*. (18. 05 2019). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/de-de/sql/t-sql/statements/alter-table-transact-sql?view=sql-server-2017>
- Microsoft SQL Dokumentation*. (11. 02. 2019). Abgerufen am 13. 08. 2019 von <https://docs.microsoft.com/de-de/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-2017>
- Microsoft SQL-Dokumentation*. (13. 08. 2019). Abgerufen am 14. 03. 2018 von <https://docs.microsoft.com/de-de/sql/relational-databases/triggers/dml-triggers?view=sql-server-2017>
- Mihalcea, V. (18. 01. 2019). *Vlad Mihalcea*. Abgerufen am 13. 08 2019 von <https://vladmihalcea.com/query-pagination-jpa-hibernate/>
- Mihalcea, V. (23. 04. 2019). *Vlad Mihalcea*. Abgerufen am 13. 08. 2019 von <https://vladmihalcea.com/the-best-way-to-map-a-onetomany-association-with-jpa-and-hibernate/>
- Mihalcea, V. (22. 01. 2019). *Vlad Mihalcea*. Abgerufen am 13. 08. 2019 von <https://vladmihalcea.com/the-best-way-to-map-a-composite-primary-key-with-jpa-and-hibernate/>
- Mihalcea, V. (19. 01. 2019). *vladmihalcea.com*. Abgerufen am 13. 08. 2019 von <https://vladmihalcea.com/the-best-way-to-map-a-many-to-many-association-with-extra-columns-when-using-jpa-and-hibernate/>
- Mutschke, P. (12 1995). *Relationale Datenbanksysteme im Vergleich*. Bonn, Deutschland: Informationszentrum Sozialwissenschaften. Von https://www.gesis.org/fileadmin/upload/forschung/publikationen/gesis_reihen/iz_arbeitsberichte/ab5.pdf abgerufen
- Oracle Help Center*. (n.d.). Abgerufen am 13. 08. 2019 von <https://docs.oracle.com/database/121/DWHSG/basicmv.htm#DWHSG8168>

- Radtke, M. (01. 02. 2019). *Big Data Insider*. Abgerufen am 13. 08. 2019 von <https://www.bigdata-insider.de/was-ist-big-data-a-562440/>
- Randal, P. S., Tripp, K., & Delaney, J. K. (03 2009). *Microsoft SQL Server 2008 Internals*. Abgerufen am 13. 08. 2019
- Rausch, C. (1999). *Zusammenfassung: Datenbanken und SQL*. Abgerufen am 13. 08. 2019 von https://ab.inf.uni-tuebingen.de/teaching/ss03/asa/db_intro.pdf
- Retama, F. M. (14. 02 2011). *Microsoft Developer*. Abgerufen am 13. 08. 2019 von <https://blogs.msdn.microsoft.com/felixmar/2011/02/14/partitioning-archiving-tables-in-sql-server-part-1-the-basics>
- Rouse, M. (11 2018). *TechTarget*. Abgerufen am 13. 08. 2019 von <https://searchdatabackup.techtarget.com/definition/data-archiving>
- Roy. (23. 04. 2010). *Royontechnology*. Abgerufen am 13. 08. 2019 von [https://royontechnology.blogspot.com/2010/04/note-on-allocation size-parameter-of.html](https://royontechnology.blogspot.com/2010/04/note-on-allocation-size-parameter-of.html)
- Rusanu Consulting. (24. 02. 2014). Abgerufen am 13. 08. 2019 von <http://rusanu.com/2014/02/24/how-to-analyse-sql-server-performance/>
- Schmidt, J. (29. 05. 2013). *Heise Online*. Abgerufen am 13. 08. 2019 von <https://www.heise.de/developer/meldung/EclipseLink-2-5-stellt-Referenzimplementierung-fuer-JPA-2-1-1872368.html>
- SQL Data Generator*. (2019). Von Redgate: <https://www.red-gate.com/products/sql-development/sql-data-generator/> 05.07.2019 abgerufen
- SQL Data Generator*. (2019). Von Redgate: <https://www.red-gate.com/products/sql-development/sql-data-generator/> 05.07.2019 abgerufen
- Stackoverflow*. (10. 06. 2010). Abgerufen am 13. 08. 2019 von <https://stackoverflow.com/questions/3019261/archive-parent-child-table-hierarchy-in-mysql?rq=1>
- Stadtarchiv-Lemgo.de*. (n.d). Abgerufen am 13. 08. 2019 von <https://www.stadtarchiv-lemgo.de/veranstaltungen-und-hinweise/kleiner-archivleitfaden-nicht-nur-fuer-schuelerinnen-und-schueler/>
- Suda, M. (31. 12. 2014). *Java-connect*. Abgerufen am 13. 08. 2019 von <http://www.java-connect.com/hibernate-tutorial/what-is-hibernate-n1-problems-and-its-solution>
- Sundar, N. (22. 02. 2018). *MSSQLTips*. Abgerufen am 13. 08. 2019 von <https://www.mssqltips.com/sqlservertip/5296/implementation-of-sliding-window-partitioning-in-sql-server-to-purge-data/>
- Tahchiev, P., Leme, F., & Massol, V. (2010). *JUnit in Action, 2. Auflage, Seite 4*. Manning Publication. doi:1935182021
- Tutorialspoint. (2019). <https://www.tutorialspoint.com/hibernate>.
- Veikko Krypczyk, C. E. (13. 09. 2018). *Entwickler.de*. Von <https://entwickler.de/online/datenbanken/grundkurs-datenbanken-datenbanksysteme-579859345.html>, abgerufen am xx abgerufen
- Vogel, L., & Scholz, S. (20. 06. 2017). *Vogella*. Abgerufen am 13. 08. 2019 von <https://www.vogella.com/tutorials/JavaPersistenceAPI/article.html>
- Vuk, A., & Geelen, P. (01. 06. 2016). *Microsoft TechNet*. Abgerufen am 13. 08. 2019 von <https://social.technet.microsoft.com/wiki/contents/articles/34477.sql-server-commands-dml-ddl-dcl-tcl.aspx>
- Wikipedia*. (19. 10. 2018). Abgerufen am 13. 08. 2019 von <https://de.wikipedia.org/wiki/Objektdatenbank>
- Wikipedia*. (24. 06. 2019). Von [https://de.wikipedia.org/wiki/Java_\(Programmiersprache\)](https://de.wikipedia.org/wiki/Java_(Programmiersprache)) abgerufen

-
- Wikipedia*. (27. 02. 2019). Abgerufen am 13. 08. 2019 von
[https://de.wikipedia.org/wiki/Annotation_\(Java\)](https://de.wikipedia.org/wiki/Annotation_(Java))
- Wikipedia.de*. (20. 09. 2018). Abgerufen am 13. 08. 2019 von
<https://de.wikipedia.org/wiki/Schneeflockenschema>
- Wikipedia.de*. (17. 05. 2019). Abgerufen am 13. 08. 2019 von
<https://de.wikipedia.org/wiki/Denormalisierung>
- Wilming, H. (01. 02 2013). *Akquinet AG*. Abgerufen am 13. 08. 2019 von <https://blog-de.akquinet.de/2013/02/01/vortragsfolien-jpa-der-persistenz-standard-in-java/>
- Yaseen, A. (17. 03. 2016). *SQLShack*. Abgerufen am 13. 08. 2019 von
<https://www.sqlshack.com/sql-server-indexed-views/>
- Yaseen, A. (12. 07. 2017). *MSSQLTips*. Abgerufen am 13. 08. 2019 von
<https://www.mssqltips.com/sqlservertip/4939/how-to-force-a-parallel-execution-plan-in-sql-server-2016/>
- Yatin. (24. 08. 2017). *JavaCodeGeeks*. Abgerufen am 13. 08. 2019 von
<https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-crud-operations-tutorial/#introduction>

Anhang

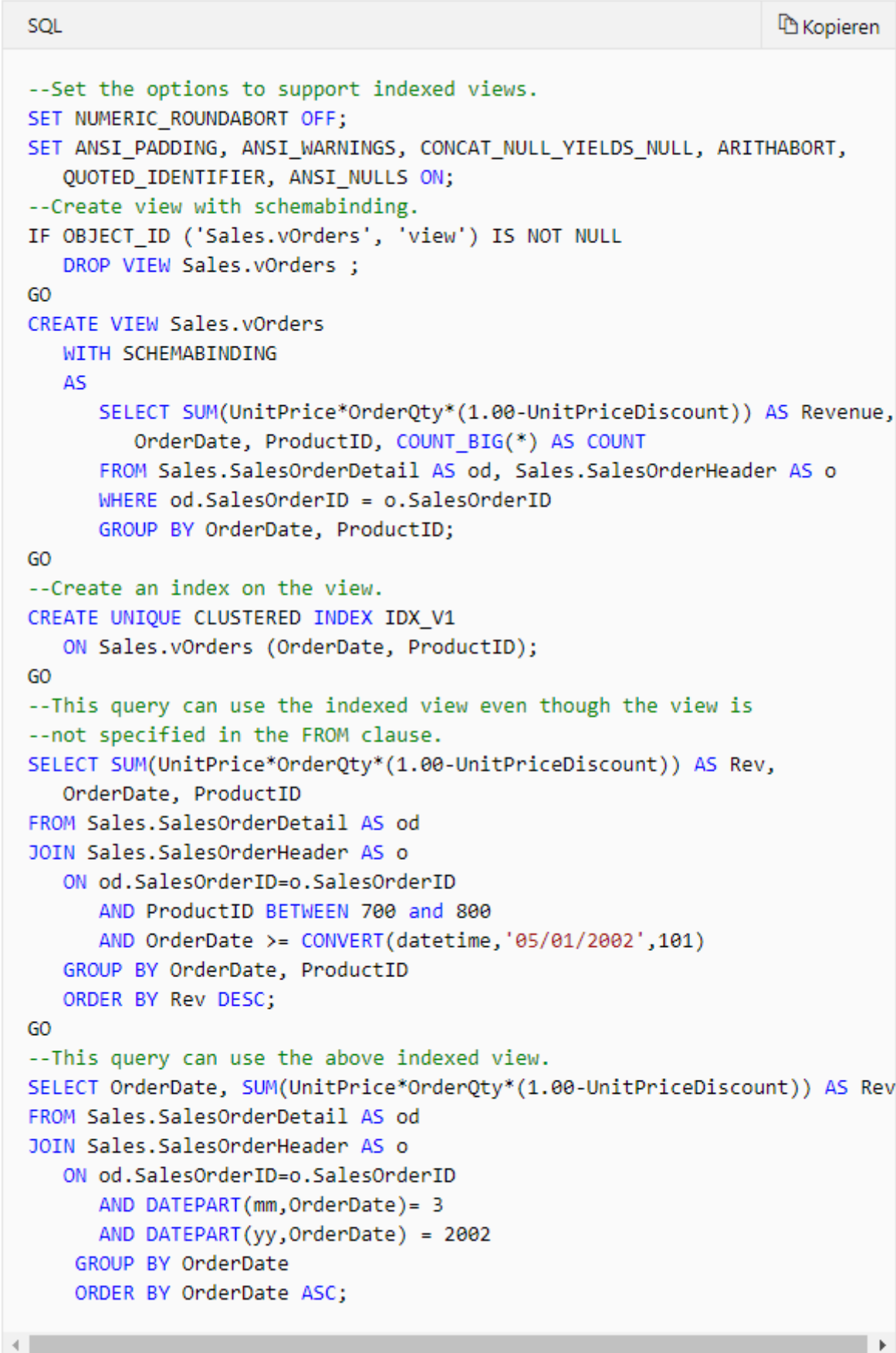
```
--Stored Procedure erstellen
Create procedure dbo.CreateNextPartition (@DtNextBoundary as
datetime)
as
Begin
Declare @DtOldestBoundary AS datetime
Declare @strFileGroupToBeUsed AS VARCHAR(100)
Declare @PartitionNumber As int

SELECT @strFileGroupToBeUsed = fg.name, @PartitionNumber =
p.partition_number, @DtOldestBoundary = cast(prv.value as datetime)
FROM sys.partitions p
INNER JOIN sys.sysobjects tab on tab.id = p.object_id
INNER JOIN sys.allocation_units au ON au.container_id = p.hobt_id
INNER JOIN sys.filegroups fg ON fg.data_space_id = au.data_space_id
INNER JOIN SYS.partition_range_values prv ON prv.boundary_id =
p.partition_number
INNER JOIN sys.partition_functions PF ON pf.function_id =
prv.function_id
WHERE 1=1
AND pf.name = 'myPartitionFunction'
AND tab.name = 'myOrderTable'
AND cast(value as datetime) = (
SELECT MIN(cast(value as datetime)) FROM sys.partitions p
INNER JOIN sys.sysobjects tab on tab.id = p.object_id
INNER JOIN SYS.partition_range_values prv ON prv.boundary_id =
p.partition_number
INNER JOIN sys.partition_functions PF ON pf.function_id =
prv.function_id
WHERE 1=1
AND pf.name = 'myPartitionFunction'
AND tab.name = 'myOrderTable'
)

Select @DtOldestBoundary Oldest_Boundary , @strFileGroupToBeUsed
FileGroupToBeUsed,@PartitionNumber PartitionNumber
Truncate Table myOrderTable with (Partitions(@PartitionNumber))
EXEC('Alter Partition Scheme myPartitionScheme NEXT USED
'+@strFileGroupToBeUsed)
Alter Partition Function myPartitionFunction() SPLIT RANGE
(@DtNextBoundary);
Alter Partition Function myPartitionFunction() MERGE RANGE
(@DtOldestBoundary);
End
Go

Execute CreateNextPartition '20030501'
Go
```

Abbildung 65: Anhang: Sliding Window gekapselt in einer Stored Procedure



```
SQL Kopieren  
  
--Set the options to support indexed views.  
SET NUMERIC_ROUNDABORT OFF;  
SET ANSI_PADDING, ANSI_WARNINGS, CONCAT_NULL_YIELDS_NULL, ARITHABORT,  
    QUOTED_IDENTIFIER, ANSI_NULLS ON;  
--Create view with schemabinding.  
IF OBJECT_ID ('Sales.vOrders', 'view') IS NOT NULL  
    DROP VIEW Sales.vOrders ;  
GO  
CREATE VIEW Sales.vOrders  
    WITH SCHEMABINDING  
    AS  
        SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Revenue,  
            OrderDate, ProductID, COUNT_BIG(*) AS COUNT  
        FROM Sales.SalesOrderDetail AS od, Sales.SalesOrderHeader AS o  
        WHERE od.SalesOrderID = o.SalesOrderID  
        GROUP BY OrderDate, ProductID;  
GO  
--Create an index on the view.  
CREATE UNIQUE CLUSTERED INDEX IDX_V1  
    ON Sales.vOrders (OrderDate, ProductID);  
GO  
--This query can use the indexed view even though the view is  
--not specified in the FROM clause.  
SELECT SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev,  
    OrderDate, ProductID  
FROM Sales.SalesOrderDetail AS od  
JOIN Sales.SalesOrderHeader AS o  
    ON od.SalesOrderID=o.SalesOrderID  
    AND ProductID BETWEEN 700 and 800  
    AND OrderDate >= CONVERT(datetime,'05/01/2002',101)  
GROUP BY OrderDate, ProductID  
ORDER BY Rev DESC;  
GO  
--This query can use the above indexed view.  
SELECT OrderDate, SUM(UnitPrice*OrderQty*(1.00-UnitPriceDiscount)) AS Rev  
FROM Sales.SalesOrderDetail AS od  
JOIN Sales.SalesOrderHeader AS o  
    ON od.SalesOrderID=o.SalesOrderID  
    AND DATEPART(mm,OrderDate)= 3  
    AND DATEPART(yy,OrderDate) = 2002  
GROUP BY OrderDate  
ORDER BY OrderDate ASC;
```

Abbildung 66: Anhang: Erstellung einer Indexed-View und Nutzung ohne explizite Nennung (Microsoft SQL Dokumentation, 2018)

Query Nr.	Abfrage
1	<code>SELECT * FROM dbo.Orders</code>
2	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUS- TOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.OR- DER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PROD- UCT_ID</code>
3	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUS- TOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.OR- DER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PROD- UCT_ID WHERE b.orderDate = '20190603 11:51:40:840'</code>
4	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount, c.quantity, d.PRODUCT_ID, d.productName, c.unitPrice FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUS- TOMER_ID INNER JOIN dbo.OrderItem c ON c.ORDER_ID = b.OR- DER_ID INNER JOIN dbo.Product d ON d.PRODUCT_ID = c.PROD- UCT_ID WHERE b.orderDate >= '20190603 11:51:40:840' AND b.orderDate <= '20190903 11:51:40:840'</code>
5	<code>SELECT a.CUSTOMER_ID, a.lastName, b.orderDate, b.totalAmount FROM Customer a INNER JOIN dbo.Orders b ON b.CUSTOMER_ID = a.CUS- TOMER_ID WHERE b.orderDate >= '20190603 11:51:40:840' AND b.orderDate <= '20190903 11:51:40:840'</code>

Tabelle 19: Anhang: Queries aus Kapitel Test und Vergleich

**Die Ergebnisse der Queries aus der Ablage Abfragen in tabellarischer Form dargestellt
inklusive der Durchschnittszeiten (AVG)**

Query Nr.	1						
Durchlauf	Datenbanken						
	db1	db1_part	db2	db2_part	db3	db3_part	
1	1438	1609	14046	14001	141298	142198	
2	1598	1472	14066	14024	140984	141098	
3	1774	1544	13893	14902	141570	141001	
4	1662	1540	14225	13823	141543	138835	
5	1521	1542	14437	13872	141908	139378	
AVG (ms)	1598,6	1541,4	14133,4	14124,4	141460,6	140502	

Query Nr.	2						
Durchlauf	Datenbanken						
	db1	db1_part	db2	db2_part	db3	db3_part	
1	7303	7081	70227	71478	7161613	7045439	
2	7387	7141	71218	70189	7194735	7133689	
3	7138	7164	72337	70768	7210934	7129805	
4	7332	7128	71879	71231	7189014	7098341	
5	7215	7138	69071	70958	7149837	7109983	
AVG (ms)	7275	7130,4	70946,4	70924,8	7181226,6	7103451,4	

Query Nr.	3						
Durchlauf	Datenbanken						
	db1	db1_part	db2	db2_part	db3	db3_part	
1	21	3	66	4	819	5	
2	24	5	96	6	867	8	
3	24	2	96	6	863	8	
4	24	5	96	6	863	8	
5	23	4	93	6	866	8	
AVG (ms)	23,2	3,8	89,4	5,6	855,6	7,4	

Query Nr.	4						
Durchlauf	Datenbanken						
	db1	db1_part	db2	db2_part	db3	db3_part	

1	1860	1838	20219	17453	168453	159802
2	2134	1827	17624	17438	183343	161023
3	1828	1808	17583	17594	188450	161189
4	1860	1843	18380	17387	177953	158991
5	1816	1871	18297	17516	180920	162894
AVG (ms)	1899,6	1837,4	18420,6	17477,6	179823,8	160779,8

Query Nr.	5					
Durschlauf	Datenbanken					
	db1	db1_part	db2	db2_part	db3	db3_part
1	466	442	4183	4172	48758	41634
2	453	422	4178	4143	42137	41597
3	426	441	4187	4130	41280	41756
4	454	446	4142	4155	41526	41089
5	458	419	4417	4023	42089	41885
AVG (ms)	451,4	434	4221,4	4124,6	43158	41592,2

Tabelle 20: Anhang: Ergebnisse in ms aus Test und Vergleich



Abbildung 67: Anhang: Durchschnittsergebnisse der Tabelle: 19 dargestellt in Diagramme

Erklärung der Kandidatin / des Kandidaten

- ☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.
- ☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist im Kapitel „Verantwortliche“ zu Beginn der Dokumentation aufgeführt.

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Name der Mitverfasser:
.....

Datum

Unterschrift der Kandidatin / des Kandidaten