

# **Der Star Schema Benchmark mit der SAP HANA Datenbank**

für die Prüfung zum

**Bachelor of Science**

des Studiengangs Angewandte Informatik International Business Competence

an der Dualen Hochschule Baden-Württemberg Mannheim

von

**Jendrik Jordan, Arwed Mett, Simon Oswald, Dominic Steinhauser**

28. März 2107

**Kurs**

**Matrikelnummern**

**Gutachter**

TINF14AIBC

1807718, 4278042, 6594512, 7344208

Prof. Dr. Colgen

# Erklärung

Wir versichern hiermit, dass wir unsere Bachelorarbeit mit dem Thema: *Der Star Schema Benchmark mit der SAP HANA Datenbank* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, 28. März 2107

---

Jendrik Jordan, Arwed Mett, Simon Oswald, Dominic Steinhauser

## **Abstract**

Die nachfolgende Arbeit betrachtet die Ausführung des Star Schema Benchmarks mit einer SAP HANA Datenbank. Grundlegend werden einige Unterschiede von verschiedenen Speicherarten, Column bzw. Row Store, in In-Memory Datenbanktabellen aufgezeigt. Insbesondere werden Laufzeitunterschiede, Verbesserungsmöglichkeiten der Laufzeit mit Indizes, sowie die Auswirkungen von verschiedenen Hardwarevoraussetzungen eines Column Store und Row Stores betrachtet und analysiert.

# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>V</b>
<b>Tabellenverzeichnis</b>	<b>VI</b>
<b>Listings</b>	<b>VII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Ziele . . . . .	1
1.2 Aufbau . . . . .	1
<b>2 Columnstore und Rowstore im Vergleich</b>	<b>2</b>
2.1 Der Columnstore . . . . .	2
2.2 Der Rowstore . . . . .	3
2.3 Rowstore und Columnstore im direkten Vergleich im Umfeld der SAP HANA	3
<b>3 Star Schema Benchmark</b>	<b>5</b>
3.1 Merkmale . . . . .	5
3.2 Datenbanklayout . . . . .	6
<b>4 Setup</b>	<b>8</b>
4.1 Virtuelle Maschine . . . . .	8
4.2 „Eclipse“ . . . . .	9
4.3 SAP HANA HDBSQL . . . . .	9
4.4 Datenbankschema . . . . .	10
4.5 Daten Import . . . . .	10
4.6 Query Execution Plan . . . . .	11
<b>5 Ausführung des Benchmarks</b>	<b>12</b>
5.1 Ziele . . . . .	12
5.2 Realisierung der Ziele . . . . .	13
5.3 Durchführung . . . . .	14
5.4 Verwendung des Benchmarks . . . . .	14
<b>6 Auswertung Benchmark</b>	<b>17</b>
6.1 Gesamtlaufzeit des Benchmarks . . . . .	17
6.2 Vergleich der SSBM Queries . . . . .	18
6.3 Vergleich der Query Execution Pläne - Query 4.3 . . . . .	21
6.4 Einfluss der grundlegenden Indizes . . . . .	25
6.5 Auswirkung unterschiedlicher Hardwarekonfiguration . . . . .	31
<b>Literatur</b>	<b>36</b>



# Abbildungsverzeichnis

5.1	Durchführung . . . . .	14
6.1	Gesamtlaufzeit von Row- und Columnstore . . . . .	18
6.2	Query-Gruppen Laufzeit . . . . .	19
6.3	Q3 Vergleich Row vs Column . . . . .	20
6.4	Q1 Vergleich Row vs Column . . . . .	20
6.5	Query Execution Plan für Q4.3 - Column Store . . . . .	22
6.6	Join - Query Execution Plan für Q4.3 - Column Store . . . . .	23
6.7	Query Execution Plan für Q4.3 - Row Store . . . . .	24
6.8	Vergleich der Gesamtlaufzeit für Column-Store mit Indizes, testweise mit OLAP-Hint. n=100 . . . . .	29
6.9	Benchmark im Columnstore unter Variation der CPU Kerne . . . . .	32
6.10	Benchmark im Columnstore unter Variation des RAMs . . . . .	33
6.11	Benchmark im Rowstore unter Variation der CPU Kerne . . . . .	34
6.12	Benchmark im Rowstore unter Variation des RAMs . . . . .	35
A1	Q3.1 Execution Plan - Comlumn Store . . . . .	42
A2	Q3.1 Execution Plan - Row Store . . . . .	43
A3	Q1.1 Execution Plan - Column Store . . . . .	44
A4	Q1.2 Execution Plan - Column Store . . . . .	45
A5	Q1.1 Execution Plan - Row Store . . . . .	46
A6	Q1.2 Execution Plan - Row Store . . . . .	47
A7	Q3.4 Execution Plan - Column Store mit Indizes . . . . .	48
A8	Q3.4 Execution Plan - Column Store ohne Indizes . . . . .	49
A9	Execution Plan: Q3.4 . . . . .	50
A10	Execution Plan: Q3.4 with Index . . . . .	51

# Tabellenverzeichnis

6.1	Gesamtlauzeiten von Row- und Columnstore in msec mit 250 Wiederholungen	17
6.2	Laufzeit: Q1-4 von Row- und Columnstore in msec mit 250 Wiederholungen	18
6.3	Vergleich Row- vs. Columnstore von Q1.1-Q4.3 in msec mit 250 Wiederholungen . . . . .	19
6.5	Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Column-Store.	25
6.7	Durchschnittslaufzeit für jede Benchmarkgruppe für Column-Store. . . . .	26
6.9	Durchschnittslaufzeit für Benchmarkgruppen 1 und 3 für Column-Store. . .	26
6.10	Durchschnittslaufzeit für Query 1.2 bei Column-Store. . . . .	28
6.11	Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Column-Store. . . . .	29
6.13	Laufzeit jeder Benchmarkgruppe für Column-Store mit Index, testweise mit OLAP-Hint. n=100 . . . . .	30
6.15	Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.	30
6.17	Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.	31
6.19	Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.	31

# Listings

4.1	Ordnerstruktur . . . . .	8
4.2	HDBSQL Befehl . . . . .	9
4.3	Beispiel SQL-Befehl: Datenimport . . . . .	11
5.1	Beispiel Benchmark . . . . .	15
5.2	Aufbau der Log Datei . . . . .	16
6.1	Benchmark Query 4.3 . . . . .	21
6.2	Ohne Index . . . . .	28
6.3	Mit Index . . . . .	28
1	Schema COLUMN Store . . . . .	37
2	Schema ROW Store . . . . .	39
3	Indizes hinzufügen . . . . .	41
4	import.sql . . . . .	41



# 1 Einleitung

Die SAP HANA Datenbank kann im Umfeld von „Data Warehouse“ als Datenbanklösung eingesetzt werden. Damit verschiedene Datenbanksysteme miteinander verglichen werden können, hilft es den Star Schema Benchmark auszuführen und festgelegte Merkmale zu vergleichen.

Im folgenden Dokument wird beschrieben, wie der allgemeine Aufbau des Benchmarks, sowie die Ausführung mit der HANA Datenbank realisiert wurde. Dabei werden die zwei Arten der Datenspeicherung (Row-Store vs. Column Store) der SAP HANA Datenbank betrachtet. Dabei sollen die Auswirkungen von Hardware-Regularien, wie z.B. die Größe des Arbeitsspeichers, beleuchtet werden. Neben den Hardware Komponenten wird ebenfalls versucht, softwaretechnische Verbesserungen, z.B. in Form von Indizes, in die Analyse mit einfließen zu lassen.

## 1.1 Ziele

Die nachfolgende Ziele sollen dabei betrachtet werden:

1. Vergleich der Ausführungszeiten des Benchmarks beim Column und Row Store
2. Vergleich der Ausführungszeiten des Benchmarks mit bzw. ohne Indizes
3. Vergleich der Ausführungszeiten des Benchmarks bei verschiedenen Hardwarevoraussetzungen

## 1.2 Aufbau

Im 2. Kapitel werden die Unterschiede von Column und Row-Store näher erläutert. Danach folgt in Kapitel 3 eine genauere Beschreibung des Star Schema Benchmarks. Das nachfolgende Kapitel 4 beschäftigt sich mit dem Grundlegenden Setup der In-Memory Datenbank, das für die in Kapitel 5 beschriebene Ausführung des Benchmarks als Grundlage benötigt wird. Das letzte Kapitel, stellt eine Auswertung der gewonnen Erkenntnisse dar.

## 2 Columnstore und Rowstore im Vergleich

Grundsätzlich gibt es zwei Möglichkeiten Daten zu organisieren: als Row- und als Columnstore. Die beispielhafte Tabelle soll die logische Referenz-Repräsentation der Daten darstellen.

Name	Alter	Geschlecht
Mark	12	m
Lisa	14	w
Torsten	6	m

### 2.1 Der Columnstore

Im Columnstore liegt eine Spalten-basierte Ordnung vor, d.h. alle Einträge einer Spalte liegen nebeneinander im Speicher. Es ergibt sich folgende physische Repräsentation der Daten im Speicher:

Mark	Lisa	Torsten	12	14	6	m	w	m
------	------	---------	----	----	---	---	---	---

Der Einsatz eines Columnstores bringt folgende Vorteile:

- **Kompression:** Durch sich oft wiederholende Werte innerhalb einer Spalte können diese effizient zusammengefasst werden. Das Ergebnis ist eine Organisation, die die selben Vorteile wie eine Indexierung bringt.
- **Große Datenmengen:** Updates, die wenige Spalten, aber dafür viel Daten umfassen, sind im Columnstore effizienter.
- **Aggregation:** Die Kompression ermöglicht auch das logische Zusammenfassen der Werte. Damit werden Suchoperationen, sowie das Zusammenfassen von Werten und vergleichbare Operationen auf Spaltenebene effizienter.

- **Parallelisierung:** Durch die Spalten-orientierte Speicherung können Datenoperationen spaltenweise parallelisiert werden.

## 2.2 Der Rowstore

Im Rowstore liegt eine Zeilen-basierte Ordnung vor, d.h. alle Einträge zu einem Datensatz liegen nebeneinander im Speicher. Es ergibt sich folgende physische Repräsentation der Daten im Speicher:

Mark	12	m	Lisa	14	w	Torsten	6	m
------	----	---	------	----	---	---------	---	---

Der Einsatz eines Rowstores bringt folgende Vorteile:

- **Neue Datensätze:** Durch die serielle Abbildung im Speicher können Datensätze problemlos angefügt werden. Ausserdem ist es so möglich Datensätze einzufügen, ohne sich um deren exakte Länge kümmern zu müssen.
- **Komplettabfragen ganzer Datensätze:** Bei der Anfrage von kompletten Datensätzen können diese durch die Serialisierung in einem Zug zurückgeliefert werden.

## 2.3 Rowstore und Columnstore im direkten Vergleich im Umfeld der SAP HANA

Allein vom Aufbau der Datenbank-Tabelle lassen sich Empfehlungen ableiten. Viele Spalten sprechen für den Columnstore, da meist nur eine kleine Menge an Spalten angesprochen wird. Ausschlaggebend für die Entscheidung zu Row- oder Columnstore ist aber vor allem die Weise, in der mit der Tabelle interagiert wird. Bei vielen kleinen Updates und regelmäßiger Erweiterung um neue Datensätze ist der Rowstore zu präferieren. Von Nachteil ist allerdings die Notwendigkeit stets den kompletten Datensatz lesen zu müssen, obwohl möglicherweise nur bestimmte Spalten benötigt werden.

Sind Anfragen vor allem abhängig von Werten einzelner Spalten, sollte der Columnstore verwendet werden. Soll der Zugriff im Rowstore trotzdem abhängig von Spaltenwerten erfolgen, so werden Indizes notwendig, welche im Columnstore durch die Kompression ebenfalls meist nicht notwendig sind. Nichtsdestotrotz werden Indizes im Columnstore ebenfalls verwendet. Kritisch werden Einfügeoperationen, die Werte einer Größe einfügen sollen, welche nicht vorgesehen war. Das festlegen fester Wert-Größen wird notwendig

für die implizite Indexierung der Datensätze. Beim Überschreiten des vorab festgelegten Wertes müssen die Daten neu organisiert werden.

SAP HANA ist für den Columnstore optimiert. Es existieren einige Features, wie das Partitionieren, welche nur unter Einsatz des Columnstores verwendet werden können. Für die bestmögliche Performance pro Datenbank-Tabelle, können diese im jeweils passenden Store abgelegt werden und später gejoint werden, was dank SAP HANA auch zwischen Row- und Columnstore funktioniert. Allerdings sind Join-Operationen innerhalb des gleichen Stores empfehlenswert für die Performance. Daher kann es sinnvoll sein oft-gejointe Daten trotz genannter Punkte im anderen Store abzulegen.

## 3 Star Schema Benchmark

Der Star Schema Benchmark (SSBM) ist eine Variation des Transaction Processing Performance Council (TPC) -H Benchmarks und hat zum Ziel die Performance von Datenbankmanagementsystemen zu messen, denen eine Entscheidungs-unterstützende Data Warehouse Applikation zugrunde liegt. Der TPC verfolgt das Ziel möglichst vergleichbare Ergebnisse zu liefern beim Einsatz bei den unterschiedlichsten Produkten. Durch seine genauen Spezifikationen wird er zum Referenz-Benchmark zwischen Produkten unterschiedlicher Hersteller.

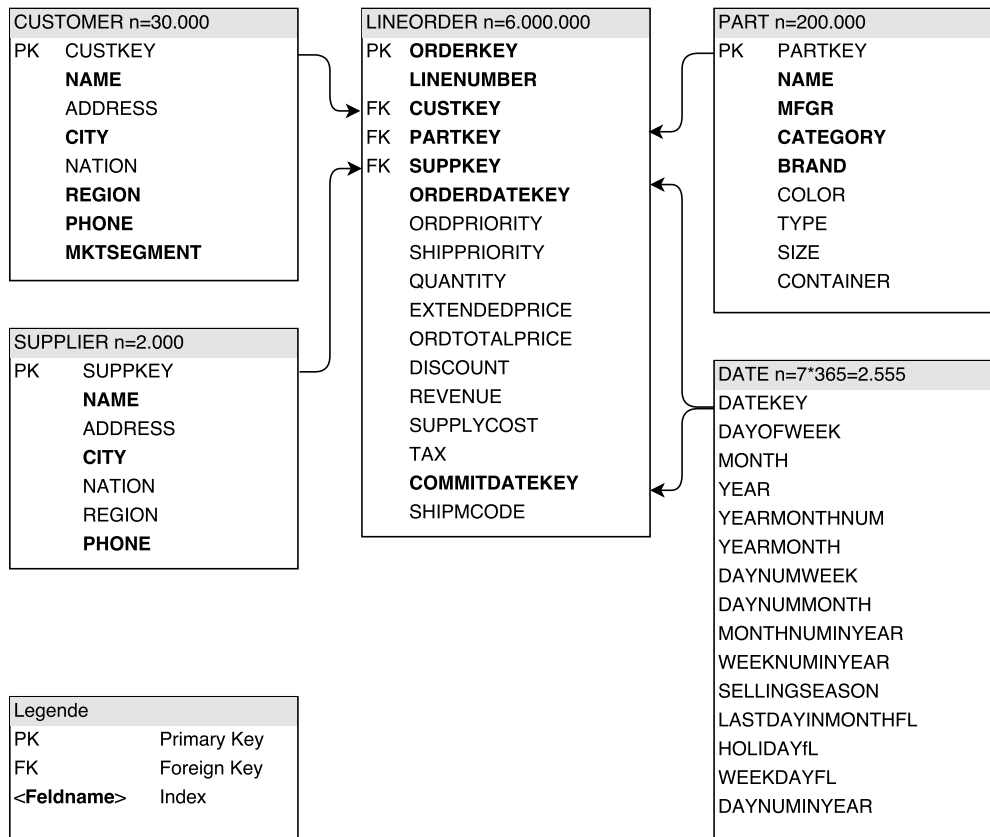
Das Ergebnis des TPC-H Benchmark ist die „TPC-H Composite Query-per-Hour“-Metrik. Durch sie werden Aspekte wie die genutzte Datenmenge oder das parallele Anfragestellen mehrerer Nutzer berücksichtigt. Damit soll sichergestellt werden, dass nur valide Vergleiche von Ergebnissen stattfinden können, da bei unterschiedlichen Voraussetzungen irreführende Ergebnisse aus dem Vergleich resultieren können.

### 3.1 Merkmale

Die Merkmale des Benchmarks sind wie folgt:

- **Ad-hoc:** Die Anfragen nutzen keinerlei Vorwissen. Hintergrund ist der Anwendungsfall von simplen GUI-gesteuerten Abfragen.
- **Ausführungsdauer:** Auf eine ausreichend große Datenmenge werden komplexe Anfragen gemacht, welche eine längere Ausführungsdauer zur Folge haben.
- **Komplexität:** Die Komplexität liegt nicht nur im Bilden der Ergebnismenge begründet, sondern auch im gleichzeitigen Ändern des Datenbestandes, sowie dem Lösen von kritischen Entscheidungen.
- **Constraints:** Eine große Menge an unterschiedlichen Constraints werden eingesetzt sowie vorausgesetzt für das operierende System, wodurch eine realitätsnahe Bedingungen geschaffen werden.
- **Relevanz:** Das Schema und die Anfragen orientieren sich an relevanten real-Anwendungsfällen.
- **Datenmenge:** Die Minimalanforderung an die Datenmenge sieht 10 Tausend Datensätze in der „SUPPLIER“-Tabelle vor, wodurch ca. 1 Gigabyte an Daten vorliegen.

## 3.2 Datenbanklayout



Das genutzte logische Schema des Benchmarks sieht folgende Tabellen vor:

- **„COSTUMER“:** Die Tabelle „COSTUMER“ hat den Primärschlüssel „CUSTKEY“ und umfasst 30 Tausend Datensätze. Indizes werden über die Felder „NAME“, „CITY“, „REGION“, „PHONE“ und „MKTSEGMENT“ gebildet.
- **„PART“:** Die Tabelle „PART“ enthält 200 Tausend Datensätze und hat den Primärschlüssel „PARTKEY“. Indizes werden über die Felder „NAME“, „MFGR“, „CATEGORY“ und „BRAND“ gebildet.
- **„SUPPLIER“:** Die Tabelle „SUPPLIER“ umfasst 2 Tausend Datensätze und hat den Primärschlüssel „SUPPKEY“. Indizes werden über die Felder „NAME“, „CITY“ und „PHONE“ gebildet.
- **„DATE“:** Die Tabelle „DATE“ hat den Primärschlüssel „DATEKEY“ und hält 2,555 Tausend Einträge. Ihre Datensätze werden über „ORDERDATEKEY“ und „COMMITDATEKEY“ zugleich aggregiert.
- **„LINEORDER“:** Die Tabelle „LINEORDER“ hat den Primärschlüssel „ORDERKEY“ und verweist über die Fremdschlüssel „CUSTKEY“, „PARTKEY“, „ORDERDATEKEY“, „COMMITDATEKEY“ und „SUPPKEY“ auf Datensätze der

anderen Tabellen. Die Tabelle umfasst 6 Millionen Einträge. Indizes werden über die Felder „ORDERKEY, „LINENUMBER, „CUSTKEY“, „PARTKEY“, „SUPPKEY“, „ORDERDATEKEY“ und „COMMITDATEKEY“ gebildet.

## 4 Setup

Dieses Kapitel beschreibt das grundlegende Setup, um mit der SAP HANA Datenbank, in Form einer virtuellen Maschine, arbeiten zu können.

### 4.1 Virtuelle Maschine

Unter dem folgenden Link<sup>1</sup> kann eine SAP HANA Instanz als virtuelle Maschine heruntergeladen werden. Für den initialen Setup, Benutzername und anlegen eines Passworts, ist dieser Link<sup>2</sup> hilfreich.

Das Tutorial beschreibt dabei, wie eine SAP HANA Datenbank mithilfe einer Virtualisierungssoftware (z.B. VMware Player oder VirtualBox) realisiert wird.

Damit der Benchmark ordnungsgemäß funktioniert, muss der virtuellen Maschine mindestens 6 GB an Arbeitsspeicher zugewiesen werden.

Um den Datenaustausch zwischen der virtuellen Maschine und dem Host bequem zu gestalten ist es nützlich, Daten, die die VM benötigt, entweder mit einem „Shared Folder“ zu teilen, oder aber mit „Secure Copy“ (SCP) in die VM zu übertragen. Dabei ist darauf zu achten, dass die Ordnerstruktur der VM und des „src“-Ordners im Github Repositories übereinstimmen. Außerdem ist es wichtig, dass die Dateien in dem Verzeichnis „/usr/sap/HXE/HDB90/work/“ abgelegt sind (vgl. Listing 4.1).

```
1 -work
2   -all_benchmarks.bash
3   -hdbsql.bash
4   -run_benchmark.bash
5   -customer.tbl
6   -date.tbl
7   -lineorder.tbl
8   -part.tbl
9   -supplier.tbl
10  -sql
11    -addBasicIndizes.sql
12    -advancedIndizes.sql
13    -bench_all.sql
14    -furtherIndizes.sql
15    -import.sql
```

<sup>1</sup> <https://www.sap.com/developer/topics/sap-hana-express.html>

<sup>2</sup> <https://www.sap.com/developer/tutorials/hxe-ua-getting-started-vm.html>



```
16 -schemaCol.sql
17 -schemaRow.sql
18 -benchmark
```

Listing 4.1: Ordnerstruktur

Zum Starten des Benchmarks muss lediglich die Datei „run\_Benchmark.bash“ gestartet werden.

## 4.2 „Eclipse“

Eclipse wird in der Regel als Entwicklungsumgebung für die SAP HANA Datenbank verwendet. Außerdem lassen sich in Eclipse Query Execution Pläne grafisch darstellen.

Unter dem folgenden Link<sup>1</sup> ist beschrieben, welche Erweiterungen und Einstellungen in „Eclipse“ vorgenommen werden müssen, um eine Verbindung zur Datenbank herzustellen. Zum Ausführen unseres Benchmarks reichen die zur Verfügung stehenden Mittel von HDBSQL vollkommen aus.

## 4.3 SAP HANA HDBSQL

SAP HANA HDBSQL ermöglicht das Ausführen von SQL Befehlen in der Kommandozeile<sup>2</sup>. Dafür muss beispielsweise der in Listing 4.2 Befehl ausgeführt werden.

```
1 hdbsql -i 90 -d SystemDB -u SYSTEM -p password -T /usr/sap/HXE/HDB90/
    work/log.log -O /usr/sap/HXE/HDB90/work/log.log -I
```

Listing 4.2: HDBSQL Befehl

### 4.3.1 Übersicht Parameter HDBSQL

- **-i**: Setzt die Instanznummer
- **-d**: Name der Datenbank
- **-u**: Name Datenbank-User

---

<sup>1</sup> <https://www.sap.com/developer/how-tos/2016/09/hxe-howto-eclipse.html>

<sup>2</sup> [http://help-legacy.sap.com/saphelp\\_hanaplatform/helpdata/en/c2/2c67c3bb571014afebeb4a76c3d95d/content.htm?frameset=/en/c2/c2/2c67c3bb571014afebeb4a76c3d95d/content.htm&current\\_toc=/en/00/0ca1e3486640ef8b884cdf1a050fbb/plain.htm&node\\_id=1155&show\\_children=false](http://help-legacy.sap.com/saphelp_hanaplatform/helpdata/en/c2/2c67c3bb571014afebeb4a76c3d95d/content.htm?frameset=/en/c2/c2/2c67c3bb571014afebeb4a76c3d95d/content.htm&current_toc=/en/00/0ca1e3486640ef8b884cdf1a050fbb/plain.htm&node_id=1155&show_children=false)

- **-p:** Passwort des Datenbank-Users
- **-T:** File in dem das Trace gespeichert wird, welches Informationen zur Ausführung des SQL Kommandos speichert.
- **-o:** Schreibt das Ergebnis der SQL Abfrage in die angegebene Datei
- **-I:** Importiert Befehle aus einer Batch Datei

## 4.4 Datenbankschema

Zu Beginn muss das Datenbankschema des SSBM-Benchmarks definiert werden. Die Tabellen werden entweder als Column Store (vgl. [Listing 1](#)) oder in einem ROW Store gespeichert (vgl. [Listing 2](#)). Die in den Listings enthaltenen „Create Table“ Statements unterscheiden sich nur anhand des Schlüsselworts „COLUMN“ bzw. „ROW“ voneinander. Damit die Auswirkungen der Indizes ebenfalls festgestellt werden können, wird der Benchmark sowohl für den „ROW“ als auch den „COLUMN“ Store zuerst ohne Indizes ausgeführt. Diese lassen sich anschließend über das Statement „CREATE Index <index\_name> ON <tabellen\_name>“ hinzufügen (vgl. [Listing 3](#)).

## 4.5 Daten Import

Der Import kann entweder über die „SQL Console“ der Entwicklungsumgebung Eclipse geschehen, oder über die Kommandozeile der virtuellen Maschine, indem mittels „HDBSQL“ die verschiedenen Dateien für das Anlegen des Schemas, den Import, etc. ausgeführt wird. Nachdem das Schema (ROW oder Column Store spielt keine Rolle) angelegt wurde, können nun die SSBM-Benchmark-Daten importiert werden. Das Importieren der Daten wird in [Listing 4](#) dargestellt. In diesem Fall werden .tbl-Daten für den Import genutzt. Es ist allerdings auch möglich die Daten über csv-Dateien in die Datenbank zu laden.

Exemplarisch ist in [Listing 4.3](#) der SQL-Befehl aufgelistet, der Daten in die Tabelle „DIM\_DATE“ importiert. Es ist ersichtlich, dass die .tbl-Datei aus dem Verzeichnis Work importiert wird. Dabei entspricht eine Zeile einem Datensatz. Die verschiedenen Attribute sind durch einen senkrechten Strich voneinander getrennt. Falls große Anzahl an Daten importiert werden müssen, so kann der Ausdruck „BATCH 1000“ dem SQL-Befehl hinzugefügt werden. Dadurch werden von der SAP HANA Datenbank mehr Ressourcen bereitgestellt (vgl. Zeile 13 [Listing 4](#) )

```
1 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/date.tbl' INTO "SYSTEM"."
   DIM_DATE"
2 WITH
3 record delimited by '\n'
4 field delimited by '|';
```

Listing 4.3: Beispiel SQL-Befehl: Datenimport

## 4.6 Query Execution Plan

Um nachvollziehen zu können, in welcher Abfolge die SQL Statements von der SAP HANA Datenbank verarbeitet werden, lassen sich mithilfe von Eclipse Query Execution Pläne visualisieren. Dafür sind folgende Schritte auszuführen:

1. SQL Console öffnen & Statement eingeben
2. „Rechtsklick“ im Context Fenster der SQL Console
3. Wähle den Menüpunkt „Visualize Plan“ → „Execute“ aus.
4. Der Query Execution Plan wird nun angezeigt.

# 5 Ausführung des Benchmarks

Das HANA Studio Plugin für Eclipse ermöglicht die direkte Ausführung von SQL-Code über die SQL-Console und würde damit ausreichen um die Schemata anzulegen, die Daten zu importieren und den Benchmark auszuführen. Werden nun jedoch Ansprüche wie das mehrfache Ausführen des Benchmarks unter unterschiedlichen Bedingungen gestellt, so ist offensichtlich, dass das händische Ausführen der einzelnen Schritte nachteilig ist. Eine elegantere Lösung ist die des Einsatzes von Skripten, welche Logik implementieren zur automatisierten Ausführung der Benchmarks. Dem Prozess der Ausführung des Benchmarks liegen dabei mehrere Gedanken zugrunde, welche in diesem Kapitel vorgestellt werden.

## 5.1 Ziele

Die erwähnten Ansprüche an den kompletten Benchmark beziehen sich unter anderem auf seine **erleichterte Ausführung**. Wie im Kapitel zum Setup vorgestellt wurde, existiert ein Bash Skript „run\_benchmark.bash“, welches das zentrale Skript darstellt und dessen alleiniger Aufruf zur Ausführung des kompletten Benchmarks ausreicht. Somit ist die Komplexität der Ausführung für den Anwender reduziert. Nicht nur wird dadurch die Ausführung an sich erleichtert, auch die Installation des HANA Studio Plugins für Eclipse wird überflüssig, da jegliche SQL-Befehle automatisch über das Skript aus SQL-Dateien ausgeführt wird. Es sind also keine SQL-Kenntnisse für den Anwender notwendig, jeglich der Umgang mit Bash-Skripten.

Dank der erleichterten Ausführung und Einrichtung der Benchmark-Umgebung ist es einfach den Benchmark auf **unterschiedlichen Test-Systemen** ausführen zu können. Durch die Variation der Test-Systeme können Faktoren wie die Anzahl zur Verfügung gestellter CPU Kerne oder RAM in ihrem Einfluss auf den Benchmark untersucht werden. Auf diese Aspekte wird im Folgekapitel näher eingegangen.

Da die Evaluierung der Ergebnisse erst durchgeführt werden kann sobald alle Ergebnisse vorhanden sind (also nach Ausführung aller Benchmarks) muss eine Möglichkeit geschaffen werden, die Ergebnisse zwischenspeichern. Zu diesem Zwecke werden während der Durchführung der Benchmarks **Logs** angelegt, welche die Daten halten.

Ein weiterer Aspekt ist die **Anzahl an Iterationen** innerhalb des kompletten Benchmarks. Viele Iterationen stellen den Ausschluss von Anomalien sicher und geben dem

**Analyser** in der späteren Evaluierung zuverlässigere Werte. Dieser wird die Ergebnisse vergleichen und in einem einzigen Dokument erfassen.

Nicht nur werden die Bedingungen für die Benchmarks variiert, sondern auch deren Inhalt. So werden unterschiedliche Konstellationen im Einsatz von **Row- und Columnstore** sowie **Indizes** durchgespielt. Genau wie die Variation des Test-Systems lassen sich dadurch essentielle Daten für den Analyser generieren.

## 5.2 Realisierung der Ziele

Die eingesetzten Test-Systeme variieren folgendermassen:

- **RAM:** Es werden 6, 8 und 12 Gigabyte von 1.6 Tausend MHz DDR3 bis hin zu 3 Tausend MHz DDR4 RAM zur Verfügung gestellt.
- **CPU:** Es werden 2, 4 und 6 virtuelle Kerne von 3.30GHz bis 4.2GHz zur Verfügung gestellt.

Die Anzahl der Iterationen wird auf den Wert 250 festgelegt.

## 5.3 Durchführung

Wie in [Abbildung 5.1](#) zu erkennen ist, lässt sich die Durchführung des Benchmarks unterteilen in die Schritte Schema Erzeugung, Daten Import, Index Erzeugung, Ausführung des Benchmarks, Speicherung der Daten ins Log und die Analyse durch den Analyser. Zur Übersichtlichkeit wird ein simplifizierter Prozess dargestellt, denn die eigentliche Durchführung involviert mehrere Unterschritte, die die angesprochene inhaltliche Variation des Benchmarks realisieren.

Nach erfolgreicher Anmeldung über die Zugangsdaten zur HANA Instanz erfolgt zuerst der Benchmark auf Basis eines Columnstores ohne Indizes. Dazu werden zuerst die Daten importiert und das Schema für den Columnstore angelegt. Anschliessend werden über den Aufruf des Skriptes „all\_benchmarks.bash“ die einzelnen Queries ausgeführt.

Nach der Ausführung des ersten Benchmarks werden in drei Schritten Indizes hinzugefügt und jeweils erneut Benchmarks durchgeführt. Darauf schliesst sich der Wechsel zum Rowstore an, was zuerst das Anlegen des Schemas für den Rowstore und ein erneutes Importieren der Daten involviert. Die Schritte zum Ausführen des Benchmarks bei unterschiedlichen Indizes werden nun wiederholt.

Eine Iteration des Skriptes resultiert damit in acht einzelnen Benchmarks. Die Daten aus den Logs werden im folgenden Schritt vom Analyser analysiert. Die Auswertung des Benchmarks bezieht sich vorallem auf folgende System-Konfiguration: 6 Kerne @ 4.2GHz bei 8 Gigabyte RAM.

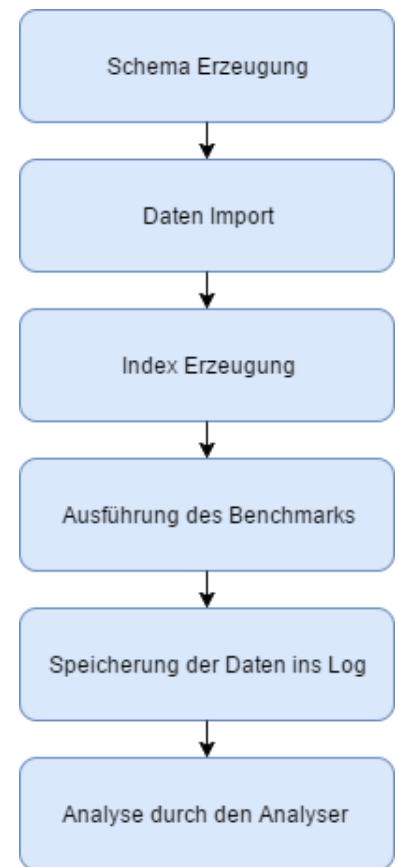


Abbildung 5.1: Durchführung

## 5.4 Verwendung des Benchmarks

Um möglichst gute Ergebnisse für den Benchmark zu erhalten ist es ein Ziel den Benchmark mehrmals auszuführen. Um dies zu realisieren war es Teil dieser Arbeit ein Script zu entwickeln welches den Benchmark automatisiert. Da die HANA Express Datenbank in einer Suse Linux basierten virtuellen Maschine betrieben werden kann, war es naheliegend dieses Script in BASH[1] zu schreiben, um keine extra Compiler oder Interpreter in der

virtuellen Maschine installieren zu müssen. Das resultierende Script kann unter [https://github.com/Osslack/HANA\\_SSBM](https://github.com/Osslack/HANA_SSBM) gefunden werden.

Durch ausführen des `run_benchmark.sh` scripts kann der Benchmark gestartet werden.

Dabei geht das Script davon aus dass die HANA Datenbank mit den Standardeinstellungen auf dem selben System installiert ist. Standardmäßig versucht das Script mittels des SYSTEM Users auf das System zuzugreifen. Um alle Einstellungen manuell zu ändern muss das Script mittels `run_benchmark.sh -v` ausgeführt werden.

Um den Benchmark anzupassen können neue Tests dem Script hinzugefügt werden. Die Mentalität dabei ist, dass der Tester dem Framework sagt, wie seine Benchmarks strukturiert sind und anschließend mittels SQL[2] und BASH spezifische Queries ausführen und wiederholen kann. Anschließend werden die Laufzeitdaten der SQL Queries als JSON[3] in einer Log Datei gespeichert. Dann kann ein Analyst mit seiner favorisierten Programmiersprache die Daten auswerten.

Dabei kann mittels des `hdb_run_file_lite` ein SQL-Script zum Setup und Cleanup der Datenbank vor bzw. nach jedem Benchmark ausgeführt werden.

Mittels `hdb_start_benchmark` kann ein neuer Benchmark gestartet werden. Dabei geht es nicht darum einen speziellen Benchmark zu starten. Es sagt lediglich, dass ein neuer Benchmark beginnt. Dadurch kann das Script das Log strukturieren. Anschließend können mehrere SQL-Kommandos mittels `all_benchmarks <path> <repetitions>` ausgeführt werden. Dabei gibt `path` an in welchem Ordner die SQL Dateien gespeichert sind. Das Skript liest alle SQL Dateien ein und führt sie „repetitions“-mal aus. Zum Beenden des Benchmarks muss `hdb_end_benchmark` aufgerufen werden. Alle Benchmarks müssen in einem `hdb_init_log - hdb_finish_log` Statement stehen damit sie in eine Log Datei geschrieben werden. Ein Beispiel ist in [Listing 5.1](#) zu sehen.

```
1 source ./hdbsql.bash
2 source ./all_benchmarks.bash
3
4 hdb_init_log
5
6 hdb_start_benchmark <benchmark name>
7 run_all_benchmarks <path> <repetitions>
8 hdb_end_benchmark
9
10 hdb_finish_log
```

Listing 5.1: Beispiel Benchmark

Ein Beispiel einer resultierenden Log Datei ist in [Listing 5.2](#) zu sehen. Es wird jeder Benchmark als Attribut gespeichert welches aus einem Array Besteht. Der Array besteht

wiederrum aus einzelnen Arrays welche die Wiederholung des Benchmarks repräsentieren. Ein einzelner Testlauf besteht aus mehreren Objekten. Jedes Objekt repräsentiert dabei einen ausgeführten Befehl. Dieser hat einen Typ welcher z.B. `exec_file` oder `inline_command` sein kann. Dieser Typ gibt an wie das Objekt zu lesen ist. So sagt zum Beispiel der Typ `exec_file` dass der Befehl durch das Ausführen einer SQL Datei gestartet wurde, wohingegen der Typ `inline_command` sagt dass der SQL Befehl direkt als String an die HANA Datenbank gesendet wurde. Dementsprechend kann ein Befehl vom Typ `exec_file` im Gegensatz zu einem Befehl des Typ `inline_command` einen Filenamen als Attribut haben.

Das Attribut `times` gibt in jedem Fall die Ausführungszeit des Befehls an. Dabei kann sich eine Query in Teilqueries unterteilen, weshalb die einzelnen Laufzeiten per Semikolon separiert in einem String gespeichert werden. Ein Analyse Werkzeug müsste die einzelnen Laufzeiten parsen und summieren um die Gesamtlaufzeit des einzelnen Befehls zu erhalten. Die Zeiten sind in Mikrosekunden angegeben.

```
1 {
2   "column_benchmark_no_index": [
3     [
4       {
5         "Type": "exec_file",
6         "Filename": "./sql/benchmark/q1_bench/q1.sql",
7         "times": " 88645;70697;39871;"
8       },
9       {
10        "Type": "exec_file",
11        "Filename": "./sql/benchmark/q1_bench/q1.1.sql",
12        "times": " 44470;"
13      }
14    ]
15  ]
16 }
```

Listing 5.2: Aufbau der Log Datei



## 6 Auswertung Benchmark

In dem vorgenommenen Benchmark wurden alle Queries 250 mal wiederholt. Im folgenden werden die Ergebnisse des Benchmarks ausgewertet. Dazu wird zu erst die Gesamtlaufzeit des Benchmarks in [Abschnitt 6.1](#) analysiert. Dabei wird zwischen zeilenbasierten und spaltenbasierten Tabellen unterschieden. Anschließend wird in [Abschnitt 6.2](#) auf die Laufzeit der einzelnen Unterabfragen des Benchmarks eingegangen. Dabei soll untersucht werden welche Abfragen besonders schnell sind. In [Abschnitt 6.4](#) und [Abschnitt 6.5](#) wird der Einfluss von Indizes bzw. unterschiedlicher Hardwarekonfigurationen untersucht.

### 6.1 Gesamtlaufzeit des Benchmarks

	Rowstore	Columnstore
Durchschnitt	371	1727
Minimum	359	1645
Maximum	488	1902
Median	369	1722
Standardabweichung	12	40
Gesamt	92855	431635

Tabelle 6.1: Gesamtlaufzeiten von Row- und Columnstore in msec mit 250 Wiederholungen

Allgemein sollten Benchmarks auf der HANA Datenbank immer zwischen Row- und Columnstore unterscheiden. Dies wird deutlich beim Betrachten der Allgemeinen Laufzeit. Wie [Tabelle 6.1](#) zu entnehmen ist, besteht ein deutlicher Unterschied in der Laufzeit zwischen Row- und Columnstore. Der Columnstore ist im Durschnitt um 79% schneller. Da der SSBM Benchmark als Maß für Abfragen im Bereich des Datawarehouse eingesetzt wird, kann also allgemein gesagt werden, dass der Columnstore dem Rowstore im Datawarehouse Umfeld vorzuziehen ist. Jedoch sollte bedacht werden, dass es sich bei dem

SSBM Benchmark um reine Abfragen von Daten handelt. Wie in [Abschnitt 2.2](#) beschrieben kann ein Rowstore von Vorteil sein, wenn Daten gespeichert werden.

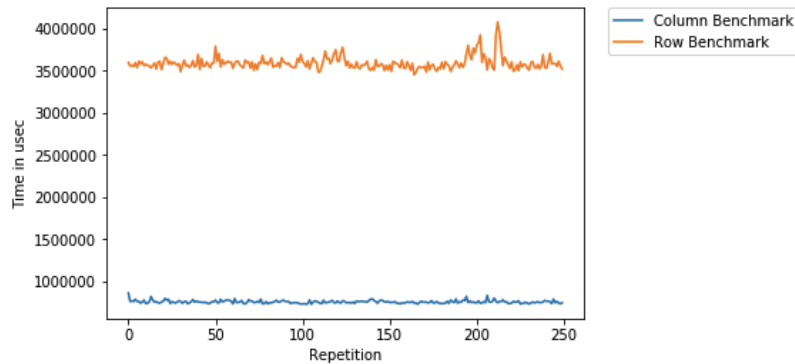


Abbildung 6.1: Gesamtlaufzeit von Row- und Columnstore

Anhand der Standardabweichung und [Abbildung 6.1](#) ist auch zu sehen, dass der Columnstore eine konstantere Zeit pro Abfrage aufweist. Allerdings da der Rowstore im allgemeinen langsamer ist als der Columnstore kann dies vernachlässigt werden, da die Standardabweichung relativ zur Gesamtlaufzeit sehr gering ist.

## 6.2 Vergleich der SSBM Queries

Im folgenden wird die Laufzeit einzelner Queries des SSBM Benchmarks separat betrachtet. Dazu wird der Benchmark in die Gruppen Q1, Q2, Q3 und Q4 unterteilt. Diese Gruppen bestehen aus einzelnen Unterabfragen Q1.1, Q1.2 etc. Zuerst wird allgemein die Geschwindigkeit der einzelnen Gruppen verglichen und anschließend auf ausgewählte Unterabfragen eingegangen.

	Columnstore				Rowstore			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
Durchschnitt	103,5	61,5	95,0	111,3	414,0	345,4	477,6	489,5
Median	101,9	60,2	93,6	109,6	412,8	343,2	474,9	485,9
Maximum	160,1	90,7	111,2	143,7	497,4	415,5	579,7	562,6
Minimum	100,3	58,3	91,4	106,3	385,6	311,1	441,5	454,0
Standardabweichung	5,7	3,9	3,5	5,0	2	15,9	17,7	17,2
Gesamt	25884,6	15385,1	23748,9	27836,4	103507,9	86354,2	119398,3	122374,7

Tabelle 6.2: Laufzeit: Q1-4 von Row- und Columnstore in msec mit 250 Wiederholungen

Wie in [Tabelle 6.2](#) zu sehen ist, ist die schnellste Query mit einer Durchschnittlaufzeit von 61,5 msec Q2 des Columnstores, wohingegen Q4 des Rowstores die langsamste Query mit einer Durchschnittlaufzeit von 489 msec ist.

Im folgenden soll verglichen werden welche Queries die größte Verbesserung durch Verwendung des Columnstores aufweisen. Dazu wird die Durchschnittlaufzeit aller einzelnen Subqueries Q1.1-Q4.3 zwischen Row- und Columnstore verglichen und der relative Performanzgewinn errechnet.

	Q1.1	Q1.2	Q1.3	Q2.1	Q2.2	Q2.3	Q3.1	Q3.2	Q3.3	Q3.4	Q4.1	Q4.2	Q4.3
Rowstore	163,56	126,25	124,69	134,93	108,39	99,14	186,91	113,75	85,76	87,23	196,88	162,72	123,76
Columnstore	35,95	47,0	21,29	28,03	25,63	7,92	31,19	23,34	21,01	20,61	47,06	42,8	22,29
Difference	127,61	79,25	103,4	106,9	82,76	91,22	155,72	90,41	64,75	66,61	149,82	119,93	101,47
in %	78,0	62,8	82,9	79,2	76,4	92,0	83,3	79,5	75,5	76,4	76,1	73,7	82,0

Tabelle 6.3: Vergleich Row- vs. Columnstore von Q1.1-Q4.3 in msec mit 250 Wiederholungen

Wie aus [Tabelle 6.3](#) zu entnehmen ist hat Q2.3 die größte Performanzsteigerung von 92 %. Im Gegensatz dazu hat Q1.2 die geringste Performanzsteigerung und liegt dabei unter der durchschnittlichen Performanzsteigerung von 78,29%.

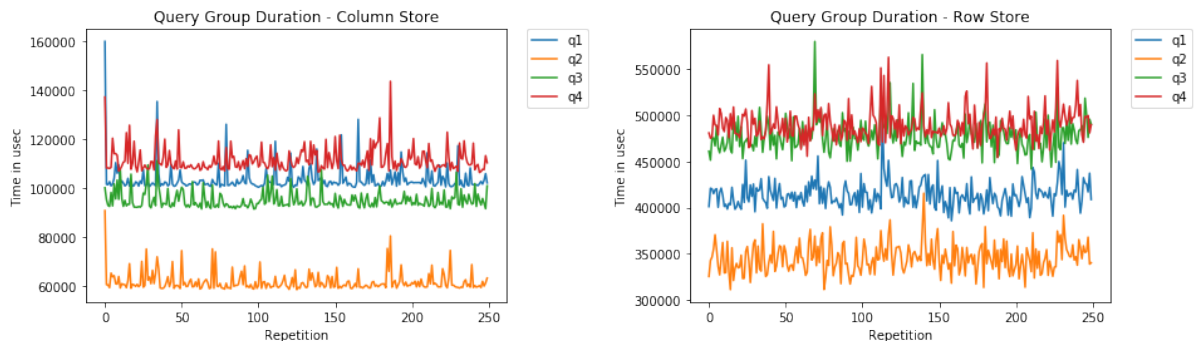


Abbildung 6.2: Query-Gruppen Laufzeit

In [Abbildung 6.2](#) ist zu erkennen, dass die Laufzeit einzelner Queries durch zeilenbasierte bzw. spaltenbasierte Tabellen beeinflusst wird. Zwar sind allgemein die spaltenorientierten Tabellen deutlich schneller, jedoch ist Q3 bei spaltenorientierten Tabellen schneller als Q1, wohingegen bei zeilenorientierten Tabellen Q1 schneller als Q3 ist. Gleich bleibt, dass Q2 die schnellsten und Q4 die langsamsten Queries sind. Um zu verstehen wodurch der Unterschied von Q1 und Q3 zustande kommt wird im folgenden der jeweils die Unterabfragen von Q1 und Q3 im Row store betrachtet. Anschließend werden die selben Queries im Row store betrachtet und dann verglichen.

Da der SSBM Benchmark so gestaltet ist dass die Queries von Q1 nach Q4 immer komplizierter werden ist es verwunderlich, dass Q3 schneller als Q1 ist. Dazu werden in [Abbildung 6.3](#) und [Abbildung 6.4](#) die Unterabfragen von Q3 betrachtet. Ziel ist es die entscheidenden Queries zu identifizieren, welche durch den Columnstore bzw. den Rowstore bevorzugt werden.

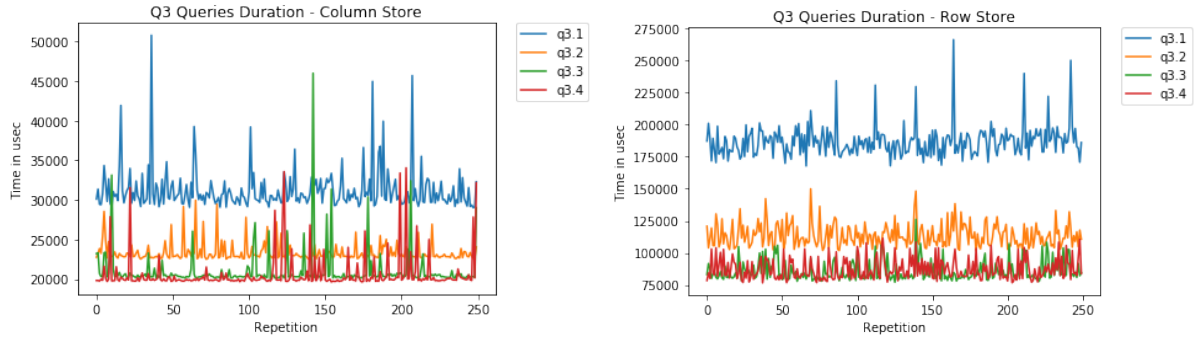


Abbildung 6.3: Q3 Vergleich Row vs Column

Aus [Abbildung 6.3](#) lässt sich erkennen, dass Q3.1 sich relativ zu Q3.2, Q3.3 und Q3.4 verbessert hat. Um dies zu verstehen werden im folgenden der Executionplan zu Q3.1 des Columnstores und des Rowstores verglichen, welche in [Abbildung A1](#) und [Abbildung A2](#) im Anhang zu finden sind. Beim Vergleich der Executionpläne wird deutlich dass der Columnstore dabei einen Filter auf 4 Tabellen durchführt. Dies ermöglicht eine Parallelisierung der Abfragen, wohingegen beim Rowstore diese Parallelisierung nicht möglich ist.

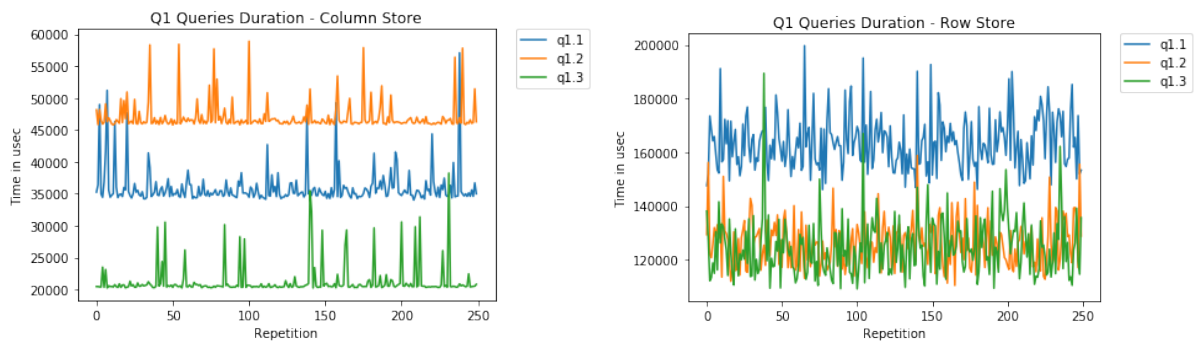


Abbildung 6.4: Q1 Vergleich Row vs Column

Interessant ist der Vergleich der Unterabfragen von Q1, da Q1.1 im Rowstore relativ zu Q1.2 langsamer ist, wohingegen im Columnstore Q1.1 schneller als Q1.3 ist. Vergleicht man die Executionpläne der Queries also Q1.1 des Columnstores mit Q1.1 des Rowstores ist keine großer Unterschied zu erkennen, welcher diesen Effekt erklären kann. Analog dazu kann beim Vergleich des Executionplans von Q1.2 des Columnstores mit dem Executionplan des Rowstores kein großer Unterschied erkannt werden. Die Executionpläne sind im

Anhang unter [Abbildung A3](#), [Abbildung A4](#), [Abbildung A5](#) und [Abbildung A6](#) zu sehen. Dementsprechend könnte der Effekt durch die interne Speicherverwaltung von HANA beeinflusst werden. Allerdings ist der Unterschied der Laufzeit beider Queries marginal.

Insgesamt kann Q3 im Rowstore besser ausgeführt werden, da eine bessere Parallelisierung möglich ist.

## 6.3 Vergleich der Query Execution Pläne - Query 4.3

In diesem Abschnitt werden die Query Execution Pläne für den Row und Column Store der SQL Abfrage 4.3, dargestellt in [Listing 6.1](#), verglichen. Ein Grund dafür sind die großen Laufzeitunterschiede zwischen Row und Column Store.

```
1 select d_year, s_city, p_brand, sum(lo_revenue - lo_supplycost) as profit
2 from lineorder
3 join dim_date on lo_orderdatekey = d_datekey
4 join customer on lo_custkey = c_customerkey
5 join supplier on lo_suppkey = s_suppkey
6 join part on lo_partkey = p_partkey
7 where
8 s_nation = 'UNITED STATES'
9 and (d_year = 1997 or d_year = 1998)
10 and p_category = 'MFGR#14'
11 group by d_year, s_city, p_brand
12 order by d_year, s_city, p_brand;
```

Listing 6.1: Benchmark Query 4.3

### 6.3.1 Query Execution Plan - Column Store

[Abbildung 6.5](#) zeigt den Ablauf der Query Execution im Column Store. Es ist zu erkennen, dass zu Beginn jede Tabelle einzeln nach den drei angegebenen Kriterien gefiltert wird.

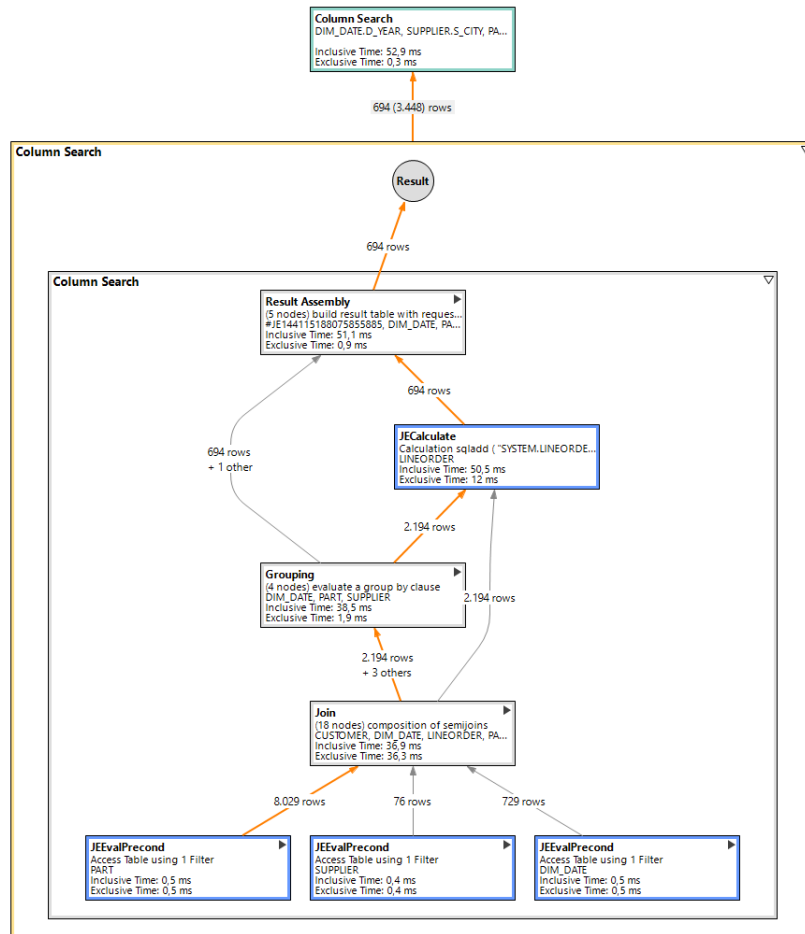


Abbildung 6.5: Query Execution Plan für Q4.3 - Column Store

Die drei Tabellen liefern dann unterschiedlich große Ergebnismengen zurück:

- Tabelle Part: 8029 Zeilen
- Tabelle SUPPLIER: 76 Zeilen
- Tabelle DIM\_DATE: 729 Zeilen

Die ermittelten Resultate werden dann in folgender Reihenfolge miteinander gejoint (vgl. [Abbildung 6.6](#)):

1. Zuerst werden die beiden Tabellen **Lineorder** und **Customer** miteinander gejoint. In diesem Join fällt auf, dass auf keine der beiden Tabellen ein Filterkriterium angewendet wird und somit der Join unabhängig ausgeführt werden kann.
2. Sobald die Ergebnismengen der Tabellen **Part**, **Supplier** und **Dim\_Date** vorliegen, werden diese ebenfalls in einer „Reduction Phase“ mit den Fremdschlüsseln der Tabelle **Lineorders** gejoint.

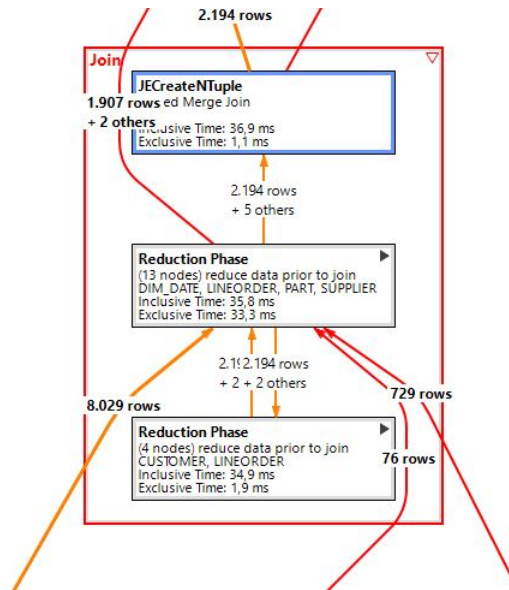


Abbildung 6.6: Join - Query Execution Plan für Q4.3 - Column Store

3. Wenn die ersten beide Schritte ausgeführt wurden, werden beide einzelnen Ergebnisse über einen Merge-Join in einer Ergebnismenge abgebildet und ausgegeben.

Anschließend an den Join wird eine Aggregation gebildet, die die Ergebnismenge nach Jahr, Nation und Produktkategorie gruppiert. Sobald die Gruppierung beendet ist, werden die Datensätze pro Gruppierung aufsteigend sortiert und anschließend ausgegeben.

### 6.3.2 Query Execution Plan - Row Store

Abbildung 6.7 zeigt den Query Execution Plan für das gleiche Query, nur dass die Daten im Row Store abgelegt sind. Ohne genaueres Hinschauen fällt auf, dass die Ausführung sich im Vergleich zum Column Store stärker an einem sequentiellen Ablauf orientiert.

Im Row Store sieht die Reihenfolge der Abarbeitung des Joins wie folgt aus:

1. Als erster Schritt wird ein Tablescan auf **Supplier** ausgeführt, der die Tabelle nach der Nation „UNITED STATES“ filtert(76 Ergebnisse).
2. Die Ergebnismenge wird mit der Tabelle **Lineorder** gejoint (228745 Ergebnisse)
3. Bevor der Hash Join ausgeführt werden kann, wird ein Index Search auf die Tabelle **PART** ausgeführt, der die Produktkategorie eingrenzt. (8029 Ergebnisse)
4. Es wird nun ein Hash Join auf mit den Ergebnisse der Schritte 1 und 2 mit den Ergebnissen des Schrittes 3 ausgeführt. Die Join wird über **LINEORDERS.LO\_Partkey** und **PART.P\_Partkey** ausgeführt. (9118 Ergebnisse)

5. Zeitgleich zu dem Hash Join kann ein Table Scan auf die Tabelle **DIM\_Date** erstellt werden. (729 Ergebnisse).
6. Anschließend werden die beiden Ergebnisse aus Schritt 4 und 5 wieder über eine Hash Join zu einem Ergebnis zusammengefasst. Dies geschieht über die Fremdschlüsselbeziehung der Tabelle **Lineorders** und **Dim\_Date** über die Spalte „LO\_Orderdatekey“ bzw. „D\_Datekey“. (2194 Ergebnisse)
7. Im letzten Join, dem Index Join, werden die zuvor in Schritt 6 ermittelten Ergebnisse mit der Tabelle **Customer** verknüpft. Dies geschieht über die Beziehung „Lineorder.LO\_Custkey = Customer.C\_Customerkey“. (2194 Ergebnisse)

Nachdem alle Joins ausgeführt wurden, wird wieder eine Aggregation gebildet und die Gruppierung anschließend aufsteigend sortiert und das Ergebnis ausgegeben.

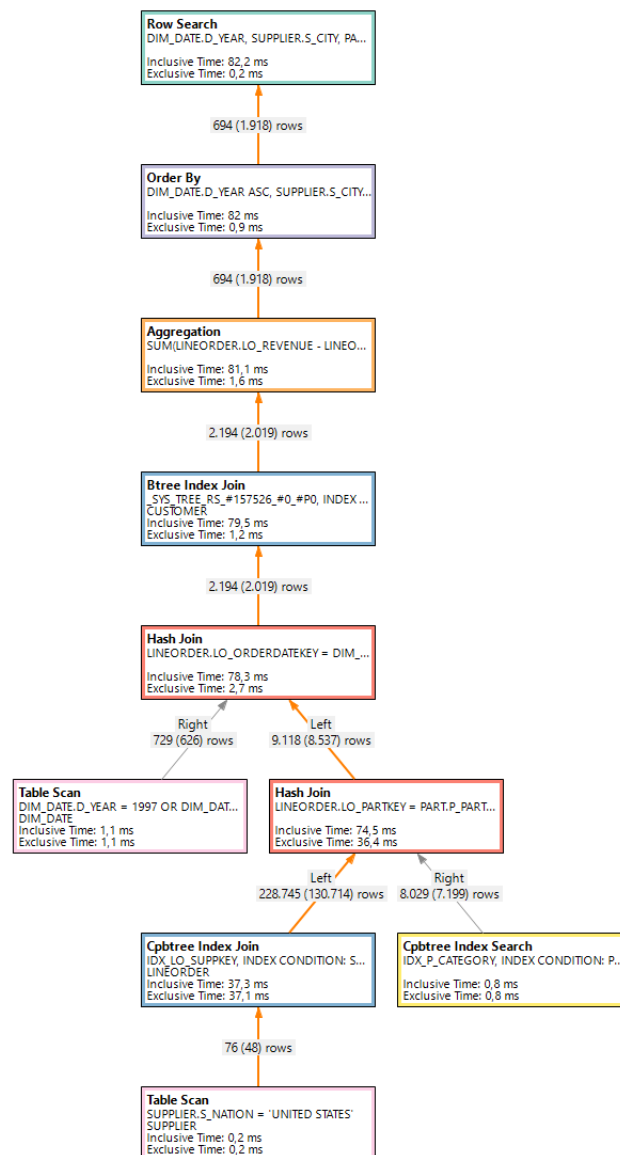


Abbildung 6.7: Query Execution Plan für Q4.3 - Row Store



### 6.3.3 Fazit Query Execution Plan

Der große zeitliche Unterschied zwischen Row und Column Store lässt sich mit der sequentielle Ausführung des Query im Row Store begründen.

In beiden Ausführungen war ersichtlich, dass die Ausführung in die 3 Schritte, Filtern, Join und Gruppieren eingeordnet werden können. Allerdings sind im Row Store das Filtern und Joinen eng miteinander verbunden und werden sogar abwechselnd nacheinander ausgeführt. Beim Column Store hingegen war die Abgrenzung zwischen Filtern und Join deutlich stärker, da zuerst die Tabellen gefiltert wurden und anschließend die Ergebnismengen miteinander verknüpft.

## 6.4 Einfluss der grundlegenden Indizes

### 6.4.1 Grundlegende Untersuchung für Column-Store

Merkmal	Col[ms]	Col Index[ms]	Abweichung[%]
Samples	250	250	
Median	752	574	-23.6%
Average	755	578	-23.4%
Total	188672	144605	-23.3%

Tabelle 6.5: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Column-Store.

Durch hinzufügen der grundlegenden Indizes wurde der Benchmark für Columstores sowohl im Schnitt als auch im Median schneller. Im Schnitt wurde er 23,4%, im Median um 23,6% schneller. Die Standardabweichung hat sich jedoch nur geringfügig verändert. Die Werte streuen relativ gesehen also gleich stark wie zuvor. Folglich konnten Mindest- und Maximallaufzeit auch deutlich reduziert werden.

Diese Ergebnisse sind interessant, da in der Regel davon ausgegangen wird, dass bei Column-Stores kein großartiges Optimierungspotenzial durch Indizes vorhanden ist. Um herauszufinden, warum trotzdem eine deutliche Verbesserung merkbar ist, wird der Query-Execution Plan des Subqueries mit der deutlichsten Verbesserung im folgenden untersucht.

**Untersuchung der Laufzeit für einzelne Query-Gruppen**

Benchmarkgruppe	Col[ms]	Col Index[ms]	Laufzeitreduzierung[ms %]
Q1	104.7	68.5	36.2   34.5%
Q2	62.1	59.7	2.4   03.8%
Q3	96.2	54.8	41.4   40.8%
Q4	112.4	106.3	6.1   05.4%

Tabelle 6.7: Durchschnittslaufzeit für jede Benchmarkgruppe für Column-Store.

Deutliche Verbesserungen sind bei den Queries der Gruppen 1 und 3 festzustellen. Hier hat sich die Laufzeit um 35% bzw. sogar 41% reduziert. Die Queries dieser Gruppe werden im Detail untersucht, um geeignete Kandidaten für die Analyse des Execution-Plans zu finden. Hier sind besonders bei Query 1.2 und 3.4 interessant, da diese die größte Verbesserung in ihrer Gruppe vorweisen können. Die Laufzeit wurde um 64% für Query 1.1 und um mehr als 90% für Query 3.4 reduziert. Woher diese Verbesserung kommen, soll im Folgenden durch die Analyse der Execution-Pläne von Query 1.1 und 3.4 geklärt werden.

Benchmark	Col[ms]	Col Index[ms]	Laufzeitreduzierung[ms %]
Q1.1	36.1	36.4	-0.3   -0.8%
Q1.2	47.8	16.8	31.0   64.8%
Q1.3	21.3	14.1	7.2   33.8%
Q3.1	31.3	31.6	-0.3   -0.9%
Q3.2	24.0	18.9	5.1   21.2%
Q3.3	21.3	2.1	19.2   90.1%
Q3.4	20.5	1.6	18.9   92.1%

Tabelle 6.9: Durchschnittslaufzeit für Benchmarkgruppen 1 und 3 für Column-Store.

### Analyse des Query-Execution-Plans für Query 3.4

**Hinweis:** Die Ergebnisse in diesem Abschnitt basieren auf Ausführung auf einem PC mit einer Intel Xeon 1230 V3 CPU mit 16GB DDR3 RAM. Die VM hatte 4 Kerne und 8GB RAM zur Verfügung.

Bei der Analyse des Query-Execution Plans zeigt sich schnell, woher die große Geschwindigkeitssteigerung kommt. Query 3.4 bildet einen Join von Lineorder auf Customer, Supplier und Dim\_Date. Dieser Join erfolgt jeweils über den Fremdschlüssel in Lineorder. Ohne Indizes ist dieser Join ausschlaggebend für die Laufzeit des Querys. Durch anlegen von Indizes auf alle Fremdschlüssel, kann der Join deutlich schneller ausgeführt werden. Den größten Vorteil hat hier der Index auf LO\_Suppkey.

Der Execution Plan ohne Indizes ist in [Abbildung A9](#), sowie [Abbildung A8](#) zu finden und der Execution Plan mit Indizes ist in [Abbildung A10](#), sowie [Abbildung A7](#) zu finden.

Query 3.1 im Vergleich nutzt zwar auch Fremdschlüssel, um einen Join zu bilden, allerdings sind hier die nicht indizierten Felder S\_Region, D\_Year, C\_Nation und S\_Nation in der Where- und der Group by-Klausel gelistet, wodurch eine Geschwindigkeitsverbesserung nicht möglich ist.

Auch bei Column-Stores scheinen sinnvoll angelegte Indizes also einen deutlichen Unterschied zu machen.

### Analyse des Query-Execution-Plans für Query 1.2

**Hinweis:** Die Ergebnisse in diesem Abschnitt basieren auf Ausführung auf einem PC mit einer Intel Xeon 1230 V3 CPU mit 16GB DDR3 RAM. Die VM hatte 4 Kerne und 8GB RAM zur Verfügung.

Bei Query 1.2 gibt es eine Besonderheit: Manchmal wird der Query über die OLAP-Engine ausgeführt<sup>1</sup>. Hierbei ist der Laufzeitunterschied zwischen Index und kein Index fast nicht mehr vorhanden, der Query mit Index braucht allerdings weniger Zeit zur Kompilation. Eigentlich ist die OLAP-Engine für „Analytical Views“, die im Star-Schema vorliegen, gedacht. Beim Star-Schema Benchmark liegen die Daten definitiv im Start Schema vor. Allerdings ist nicht klar, warum die OLAP-Engine manchmal verwendet wird und manchmal nicht. Ein Muster war hier nicht zu erkennen.

---

<sup>1</sup> Um zu sehen, welche Engine verwendet wird, muss anstatt „Visualize Plan“ die Option „Explain Plan“ gewählt werden.

Zunächst trat die OLAP-Engine nur auf, wenn Indizes hinzugefügt wurden, weshalb zunächst angenommen wurde, dass HANA an Hand der Indizes erkennt, dass es sich im Grunde um einen Analytical View handelt und dementsprechend optimiert. Allerdings gab es auch Fälle, wo die OLAP-Engine auch ohne Indizes verwendet wurde. Kommt die OLAP-Engine zum Einsatz, so ist der Query in etwa so schnell, wie mit der Column-Engine in Kombination mit Indizes, siehe [Tabelle 6.10](#).

Engine	Col [ms]	Col Index [ms]
OLAP	6	6
Column	16	6

Tabelle 6.10: Durchschnittslaufzeit für Query 1.2 bei Column-Store.

Kommt nicht die OLAP-Engine, sondern die „normale“ Column-Engine zum Einsatz, so gibt es einen deutlichen Unterschied zwischen Index und kein Index. Hierbei kann der Index auf LO\_OrderDateKey für den JOIN genutzt werden und beschleunigt diesen somit. Außerdem wird die Berechnung `sum(lo_extendedprice*lo_discount)` deutlich beschleunigt. Warum ist allerdings nicht klar, denn auf diese Felder wurde kein Index angelegt.

Schaut man sich genauer an, was an dieser Stelle passiert, so werden die gleichen Operationen auf die gleichen Datenmengen angewandt, allerdings sind die Operationen **mit** Index deutlich schneller.<sup>1</sup>

```

1 <executePop(
2   <lockInputs(num=3,)=0.00>
3   <calculateOnAttr(
4     <calculateWithAggregation(rows=4301,inputs=2,outputs=1,)=8.04>
5     rows=4301,outputs=1,)
6   =8.15>
7 )=11.61>

```

Listing 6.2: Ohne Index

```

1 <executePop(
2   <lockInputs(num=3,)=0.00>
3   <calculateOnAttr(
4     <calculateWithAggregation(rows=4301,inputs=2,outputs=1,)=1.03>
5     rows=4301,outputs=1,)
6   =1.10>
7 )=1.18>

```

Listing 6.3: Mit Index

<sup>1</sup> Die letzte Zahl scheint jeweils die Laufzeit in ms zu sein, aber eine genaue Erklärung dieser Werte war leider nicht zu finden.

Wie in Tabelle 6.11 zu sehen, ist die OLAP-Engine insgesamt sowohl mit, als auch ohne Index deutlich schneller, als die Column-Engine.

Durch den HINT „USE\_OLAP\_PLAN“ kann die OLAP-Engine als bevorzugte Engine festgelegt werden. Führt man jeden der Querys mit diesem Hint durch, so liefert dies die folgenden Ergebnisse:

Engine	No Index [ms]	Index [ms]
OLAP	187	188
Colum	296	226

Tabelle 6.11: Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Column-Store.

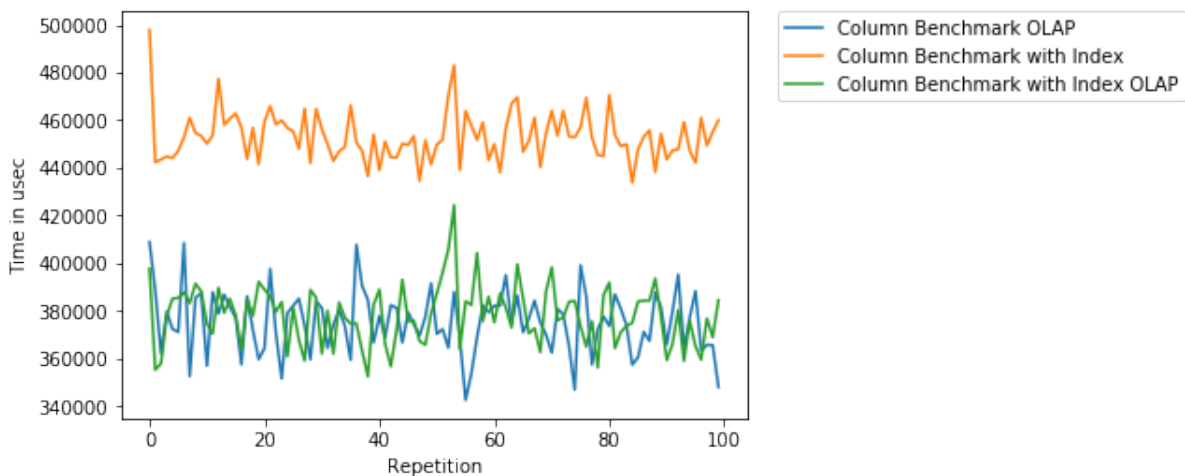


Abbildung 6.8: Vergleich der Gesamtlaufzeit für Column-Store mit Indizes, testweise mit OLAP-Hint. n=100

Wie in Grafik 6.8 zu sehen, wird durch den OLAP-Hint eine deutliche Beschleunigung erzielt. Ob zusätzlich noch ein Index existiert hat jedoch wenig bis keinen Einfluss.

Bei genauerer Betrachtung der Laufzeit pro Benchmarkgruppe fällt besonders auf, dass die Querys der Gruppe 1 nicht von der OLAP-Engine profitieren, sondern sogar langsamer werden. Warum dies beim Durchführen des Benchmarks, aber nicht bei Analyse der Execution-Pläne der Fall ist, ist nicht klar.

Die anderen Benchmarkgruppen werden durch den OLAP-Hint jedoch schneller, Gruppe 3 und 4 nur geringfügig, Gruppe 2 jedoch wird nahezu doppelt so schnell.

Wert	OLAP-Hint	Q1	Q2	Q3	Q4
Average	Nein	23.5	72.5	60.4	69.9
Average	Ja	26.3	36.4	58.3	65.6
Median	Nein	23.4	72.1	60.2	69.3
Median	Ja	26.5	36.0	59.5	66.0

Tabelle 6.13: Laufzeit jeder Benchmarkgruppe für Column-Store mit Index, testweise mit OLAP-Hint. n=100

### Fazit für Column-Stores

Auch Column-Stores können durch geschickt gewählte Indizes deutlich beschleunigt werden. Durch Nutzung der OLAP-Engine können diese Beschleunigungen jedoch nochmals teils deutlich überboten werden. Es erscheint sinnvoller, sein Augenmerk darauf zu legen, dass Querys diese auch nutzen. Dies ist zwar über einen HINT möglich, davon wird in der Praxis jedoch abgeraten.

### 6.4.2 Grundlegende Untersuchung für Row-Store

Wert	Row[ms]	Row Index[ms]	Abweichung [%]
Samples	250	250	
Median	3519	3006	14.5%
Average	3539	3048	13.8%
Total	884960	762128	

Tabelle 6.15: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.

Wert	Row[ms]	Row Index[ms]	Abweichung [%]
Samples	250	250	
Median	3426	2877	
Average	3440	2881	

Tabelle 6.17: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.

	q1.1	q1.2	q1.3	q2.1	q2.2	q2.3
Row[ms]	163	133	127	139	111	102
Row Index[ms]	156	13	3	73	16	4
Abweichung[ms]	7	120	124	66	95	98
Abweichung[%]	4.2	90.2	97.6	47.4	85.5	96.0

Tabelle 6.19: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Row-Store.

## 6.5 Auswirkung unterschiedlicher Hardwarekonfiguration

Nicht nur die Betrachtung unterschiedlicher Konfigurationen auf Software-Ebene ist interessant, sondern auch die auf simulierter Hardware-Ebene. Die eingesetzten Hardwarekonstellationen wurden im Kapitel zur Ausführung des Benchmarks beschrieben und werden nun miteinander verglichen. Zum Einsatz kommen die Ergebnisse des Analysers, welche als Visualisierungen sowie Aggregaten-Werten präsentiert werden. In der Analyse werden wird zuerst auf den Columnstore eingegangen und anschließend vergleichend auf den Rowstore.

### 6.5.1 Columnstore

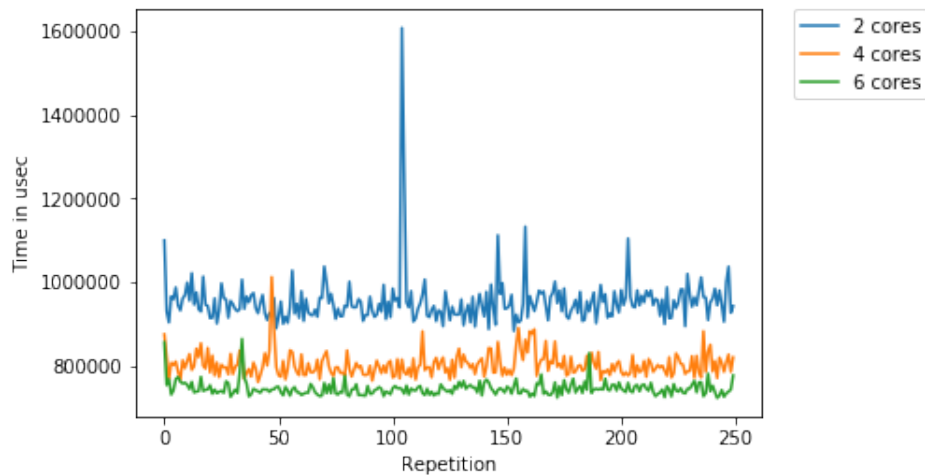


Abbildung 6.9: Benchmark im Columnstore unter Variation der CPU Kerne

Abbildung 6.9 zeigt die Ausführungsdauer über den 250 Durchläufen des Benchmarks bei konstanten 8 Gigabyte RAM. Generell bewegt sich die Ausführungsdauer pro Benchmark hauptsächlich im Bereich von ca. 0.7 Sekunden bis 1 Sekunde. Dabei werden die schnelleren Werte, wie erwartet, durch den sechs-Kerner gebildet und die langsameren Werte durch den zwei-Kerner. Werden der sechs-Kerner (0.75 Sekunden) und der zwei-Kerner (0.95 Sekunden) anhand ihren arithmetischen Mitteln miteinander verglichen so ergibt sich eine prozentuale Differenz von 26.7%. Der vier-Kerner hingegen erreicht eine durchschnittliche Ausführungszeit von 0,8 Sekunden was ihm eine Differenz von 7% im Vergleich zum sechs-Kerner einbringt. Grundsätzlich gilt, dass im Columnstore im HANA Umfeld dank der begünstigten Parallelisierung eine bessere Ausnutzung der CPU durch mehrere Threads stattfindet. Dadurch wird die Nutzung der Hardware optimiert und eine bestmögliche Performance wird erzielt. Pro zusätzlichem Kern kann die Ausführungszeit halbiert werden, indem die Last gleichmäßig auf die Kerne aufgeteilt wird. Diese Überlegung stimmt mit den Messwerten überein und es lässt sich auf einen quadratischen Zuwachs der Ausführungszeit abhängig zur Anzahl der CPU-Kerne schließen.



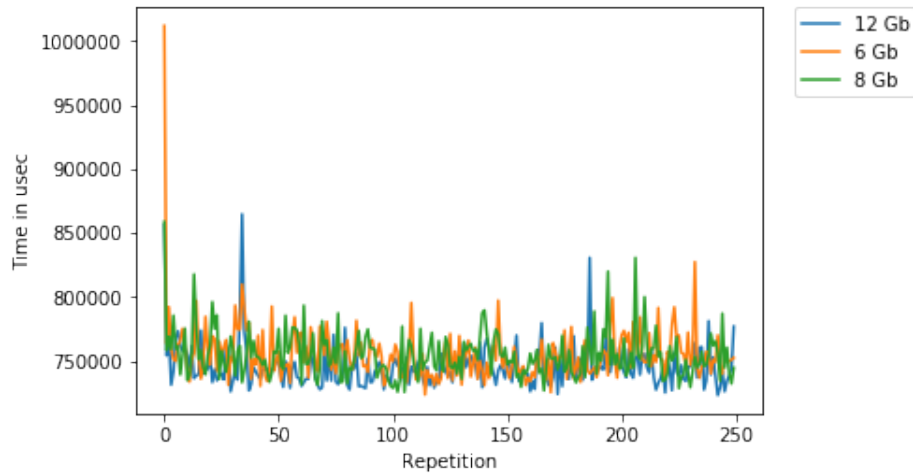


Abbildung 6.10: Benchmark im Columnstore unter Variation des RAMs

Abbildung 6.10 zeigt die Ausführungsdauer über den 250 Durchläufen des Benchmarks bei konstanten 6 CPU Kernen. Die Ausführungsdauer pro Benchmark scheint nicht markant zu variieren, auch wenn die genauere Betrachtung der Aggregat-Werte zeigt, dass der Benchmark unter 12 Gigabyte RAM im Schnitt um eine Hunderdstel Sekunde schneller ist, als bei 6 und 8 Gigabyte. Ein besseres Mittel zum Vergleich als die Durchschnittslaufzeit bildet in diesem Falle die Standardabweichung der Laufzeiten, welche ein Maß für die Stabilität des Benchmarks darstellt. Diese beträgt bei 12 Gigabyte RAM gerade einmal 16 Millisekunden, während sich bei 8 Gigabyte ein 4 prozentiger Zuwachs und bei 6 Gigabyte ein 37 prozentiger Zuwachs erkennen lässt. Daraus lässt sich ableiten, dass die Stabilität stark unter der Einschränkung des RAMs leidet, und somit einzelne Anfragen erheblich länger dauern können. Vermutlich liegt die Unregelmäßigkeit vor allem darin begründet, dass bei reduziertem RAM externe Faktoren (Cache-Miss, Zugriffszeit) deutlicher zum tragen kommen.

### 6.5.2 Rowstore

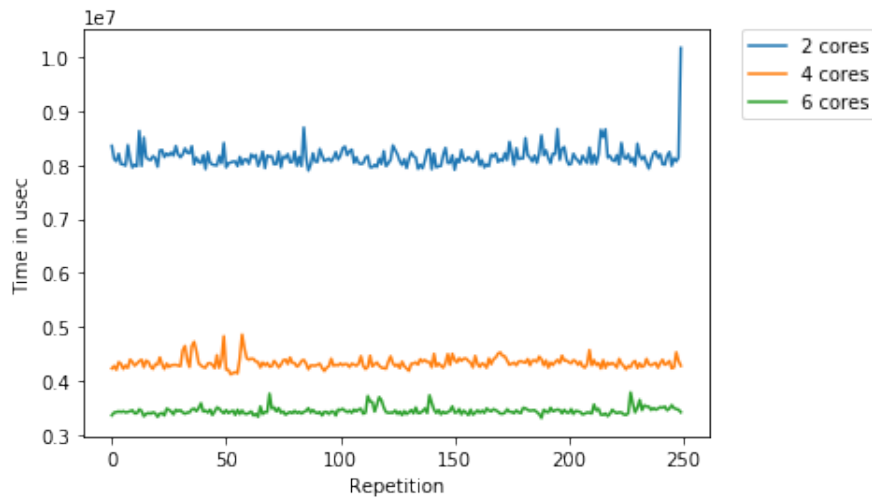


Abbildung 6.11: Benchmark im Rowstore unter Variation der CPU Kerne

Abbildung 6.11 zeigt die Ausführungsdauer über den 250 Durchläufen des Benchmarks bei konstanten 8 Gigabyte RAM. Zu erkennen ist die starke Abweichung der Ausführungszeiten, welcher deutlicher ausfällt als im Columnstore. Der zwei-Kerner liegt mit einer durchschnittlichen Ausführungszeit von 8,2 Sekunden 137% über der Ausführungszeit des acht-Kerners, welcher nur 3,4 Sekunden im Schnitt braucht. Die Abweichung des sechs-Kerners beträgt dagegen nur 26%. Es scheint als würde die Anzahl der CPU-Kerne einen größeren Einfluss haben im Rowstore als im Columnstore. Eine mögliche Erklärung dafür kann folgendermaßen aussehen: Da die Daten im Columnstore blockweise gelesen werden können, haben die Lesezugriffe eine längere Ausführungszeit, da auch viele Daten in einem Zuge gelesen werden können. Der komplette Satz an Daten wird anschliessend von der CPU ausgewertet, wobei die Auswertung kürzer dauert als der Speicherzugriff. Dadurch wird das RAM zum Bottleneck und die CPU unrelevanter für die Ausführungszeit. Im Rowstore müssen viele einzelne Lesezugriffe durchgeführt werden, wobei die einzelnen Daten direkt verarbeitet werden von der CPU. Die Parallelität ist nicht in dem Maße gegeben wie im Columnstore, wodurch stets das RAM auf die CPU wartet und umgekehrt. Daraus entsteht eine erhöhte Relevanz der CPU für die Ausführungszeit als beim Columnstore.

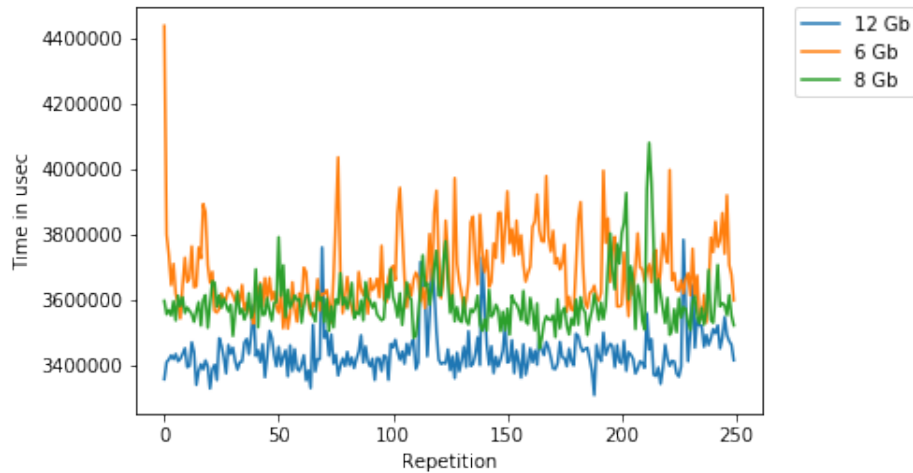


Abbildung 6.12: Benchmark im Rowstore unter Variation des RAMs

Abbildung 6.12 zeigt die Ausführungsdauer über den 250 Durchläufen des Benchmarks bei konstanten 6 CPU Kernen. Die Ausführungszeiten unterscheiden sich ebenfalls deutlicher als beim Columnstore. Unterschieden sich im Columnstore die durchschnittlichen Ausführungszeiten von 12 Gigabyte und 6 Gigabyte RAM noch um 1,33%, so liegt der Unterschied nun bereits bei 7,13%. Die Standardabweichungen scheinen dasselbe Muster wie im Columnstore aufzuweisen: Je mehr RAM, desto stabiler läuft der Benchmark. Im Falle des 6 Gigabyte Systems sind enorme Ausschläge zu beobachten, obwohl teilweise sogar die Ausführungszeit des 8 Gigabyte Systems unterboten wird. Die erhöhte Relevanz der Menge an verfügbarem RAM scheint im Rowstore ebenfalls von größerer Bedeutung für die Laufzeit zu sein als im Columnstore. Während im Columnstore automatisch Optimierungen, wie Indexierung und Kompression stattfinden, können diese im Rowstore womöglich nur bei ausreichend RAM durchgeführt werden.

# Literatur

- [1] *GNU Bash*. 2017. URL: <https://www.gnu.org/software/bash/>.
- [2] *SQL:2011*. Standard. Geneva, CH: International Organization for Standardization, 2011.
- [3] *The JavaScript Object Notation (JSON) Data Interchange Format*. Standard. Internet Engineering Task Force, 2014.

# Anhang

```
1 DROP TABLE customer;
2
3 CREATE COLUMN TABLE customer (
4     C_CUSTOMERKEY INTEGER,
5     C_Name varchar(25),
6     C_Address varchar(25),
7     C_City varchar(10),
8     C_Nation varchar(15),
9     C_Region varchar(12),
10    C_Phone varchar(15),
11    C_MktSegment varchar(10),
12    PRIMARY KEY ("C_CUSTOMERKEY")
13 );
14
15 DROP TABLE part;
16
17 CREATE COLUMN TABLE part
18 (
19     P_PartKey integer,
20     P_Name varchar(25),
21     P_MFGR varchar(10),
22     P_Category varchar(10),
23     P_Brand varchar(15),
24     P_Colour varchar(15),
25     P_Type varchar(25),
26     P_Size tinyint,
27     P_Container char(10),
28     PRIMARY KEY (P_PartKey)
29 );
30
31
32
33 DROP TABLE supplier;
34
35 CREATE COLUMN TABLE supplier (
36     S_SuppKey integer,
37     S_Name char(25),
38     S_Address varchar(25),
39     S_City char(10),
40     S_Nation char(15),
41     S_Region char(12),
```

```
42     S_Phone char(15),
43     PRIMARY KEY(S_SuppKey)
44 );
45
46
47
48 DROP TABLE dim_date;
49
50 CREATE COLUMN TABLE dim_date
51 (
52     D_DateKey integer,
53     D_Date char(18),
54     D_DayOfWeek char(9),
55     D_Month char(9),
56     D_Year smallint,
57     D_YearMonthNum integer,
58     D_YearMonth char(7),
59     D_DayNumInWeek tinyint,
60     D_DayNumInMonth tinyint,
61     D_DayNumInYear smallint,
62     D_MonthNumInYear tinyint,
63     D_WeekNumInYear tinyint,
64     D_SellingSeason char(12),
65     D_LastDayInWeekFl tinyint,
66     D_LastDayInMonthFl tinyint,
67     D_HolidayFl tinyint,
68     D_WeekDayFl tinyint,
69     PRIMARY KEY(D_DateKey)
70 );
71
72 DROP TABLE lineorder;
73
74 CREATE COLUMN TABLE lineorder
75 (
76     LO_OrderKey bigint not null,
77     LO_LineNumber tinyint not null,
78     LO_CustKey int not null,
79     LO_PartKey int not null,
80     LO_SuppKey int not null,
81     LO_OrderDateKey int not null,
82     LO_OrderPriority varchar(15),
83     LO_ShipPriority char(1),
84     LO_Quantity tinyint,
85     LO_ExtendedPrice decimal,
86     LO_OrdTotalPrice decimal,
87     LO_Discount decimal,
88     LO_Revenue decimal,
```

```
89  LO_SupplyCost decimal,  
90  LO_Tax tinyint,  
91  LO_CommitDateKey integer not null,  
92  LO_ShipMode varchar(10)  
93 );
```

Listing 1: Schema COLUMN Store

```
1 DROP TABLE  customer;  
2  
3 CREATE ROW TABLE  customer (  
4   C_CUSTOMERKEY INTEGER,  
5   C_Name varchar(25),  
6   C_Address varchar(25),  
7   C_City varchar(10),  
8   C_Nation varchar(15),  
9   C_Region varchar(12),  
10  C_Phone varchar(15),  
11  C_MktSegment varchar(10),  
12  PRIMARY KEY ("C_CUSTOMERKEY")  
13 );  
14  
15 DROP TABLE  part;  
16  
17 CREATE ROW TABLE  part  
18 (  
19   P_PartKey integer,  
20   P_Name varchar(25),  
21   P_MFGR varchar(10),  
22   P_Category varchar(10),  
23   P_Brand varchar(15),  
24   P_Colour varchar(15),  
25   P_Type varchar(25),  
26   P_Size tinyint,  
27   P_Container char(10),  
28   PRIMARY KEY (P_PartKey)  
29 );  
30  
31  
32  
33 DROP TABLE  supplier;  
34  
35 CREATE ROW TABLE  supplier (  
36   S_SuppKey integer,  
37   S_Name char(25),  
38   S_Address varchar(25),  
39   S_City char(10),  
40   S_Nation char(15),
```

```
41  S_Region char(12),
42  S_Phone char(15),
43  PRIMARY KEY(S_SuppKey)
44 );
45
46
47
48 DROP TABLE dim_date;
49
50 CREATE ROW TABLE dim_date
51 (
52  D_DateKey integer,
53  D_Date char(18),
54  D_DayOfWeek char(9),
55  D_Month char(9),
56  D_Year smallint,
57  D_YearMonthNum integer,
58  D_YearMonth char(7),
59  D_DayNumInWeek tinyint,
60  D_DayNumInMonth tinyint,
61  D_DayNumInYear smallint,
62  D_MonthNumInYear tinyint,
63  D_WeekNumInYear tinyint,
64  D_SellingSeason char(12),
65  D_LastDayInWeekFl tinyint,
66  D_LastDayInMonthFl tinyint,
67  D_HolidayFl tinyint,
68  D_WeekDayFl tinyint,
69  PRIMARY KEY(D_DateKey)
70 );
71
72 DROP TABLE lineorder;
73
74 CREATE ROW TABLE lineorder
75 (
76  LO_OrderKey bigint not null,
77  LO_LineNumber tinyint not null,
78  LO_CustKey int not null,
79  LO_PartKey int not null,
80  LO_SuppKey int not null,
81  LO_OrderDateKey int not null,
82  LO_OrderPriority varchar(15),
83  LO_ShipPriority char(1),
84  LO_Quantity tinyint,
85  LO_ExtendedPrice decimal,
86  LO_OrdTotalPrice decimal,
87  LO_Discount decimal,
```



```

88  LO_Revenue decimal,
89  LO_SupplyCost decimal,
90  LO_Tax tinyint,
91  LO_CommitDateKey integer not null,
92  LO_ShipMode varchar(10)
93 );

```

Listing 2: Schema ROW Store

```

1 CREATE INDEX idx_c_name ON customer(C_Name);
2 CREATE INDEX idx_c_city ON customer(C_City);
3 CREATE INDEX idx_c_region ON customer(C_Region);
4 CREATE INDEX idx_c_phone ON customer(C_Phone);
5 CREATE INDEX idx_c_mktsegment ON customer(C_MktSegment);
6
7 CREATE INDEX idx_p_name ON part(P_Name);
8 CREATE INDEX idx_p_mfgr ON part(P_MFGR);
9 CREATE INDEX idx_p_category ON part(P_Category);
10 CREATE INDEX idx_p_brand ON part(P_Brand);
11
12 CREATE INDEX idx_s_city ON supplier(S_City);
13 CREATE INDEX idx_s_name ON supplier(S_Name);
14 CREATE INDEX idx_s_phone ON supplier(S_Phone);
15
16 CREATE INDEX idx_lo_orderkey_lo_linenum ON lineorder(LO_OrderKey,
    LO_LineNumber);
17 CREATE INDEX idx_lo_custkey ON lineorder(LO_CustKey);
18 CREATE INDEX idx_lo_suppkey ON lineorder(LO_SuppKey);
19 CREATE INDEX idx_lo_partkey ON lineorder(LO_PartKey);
20 CREATE INDEX idx_lo_orderdatekey ON lineorder(LO_OrderDateKey);
21 CREATE INDEX idx_lo_commitdatekey ON lineorder(LO_CommitDateKey);

```

Listing 3: Indizes hinzufügen

```

1 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/date.tbl' INTO "SYSTEM"."
    DIM_DATE"
2 WITH
3 record delimited by '\n'
4 field delimited by '|';
5
6 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/customer.tbl' INTO "SYSTEM
    "."CUSTOMER"
7 WITH
8 record delimited by '\n'
9 field delimited by '|';
10
11 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/lineorder.tbl' INTO "
    SYSTEM"."LINEORDER"

```

```

12 WITH
13 batch 10000
14 record delimited by '\n'
15 field delimited by '|';
16
17 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/part.tbl' INTO "SYSTEM"."
    PART"
18 WITH
19 record delimited by '\n'
20 field delimited by '|';
21
22 IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/supplier.tbl' INTO "SYSTEM
    ". "SUPPLIER"
23 WITH
24 record delimited by '\n'
25 field delimited by '|';

```

Listing 4: import.sql

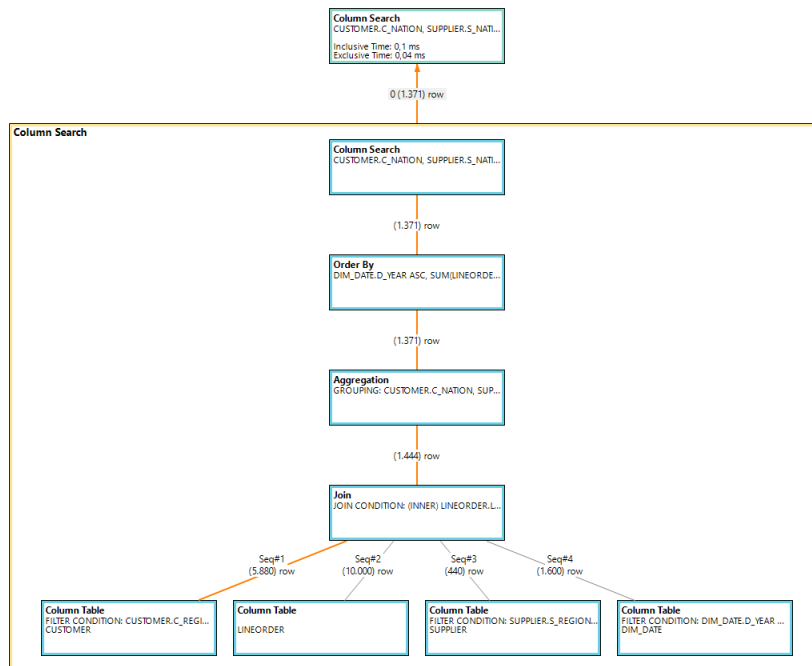


Abbildung A1: Q3.1 Execution Plan - Comlumn Store

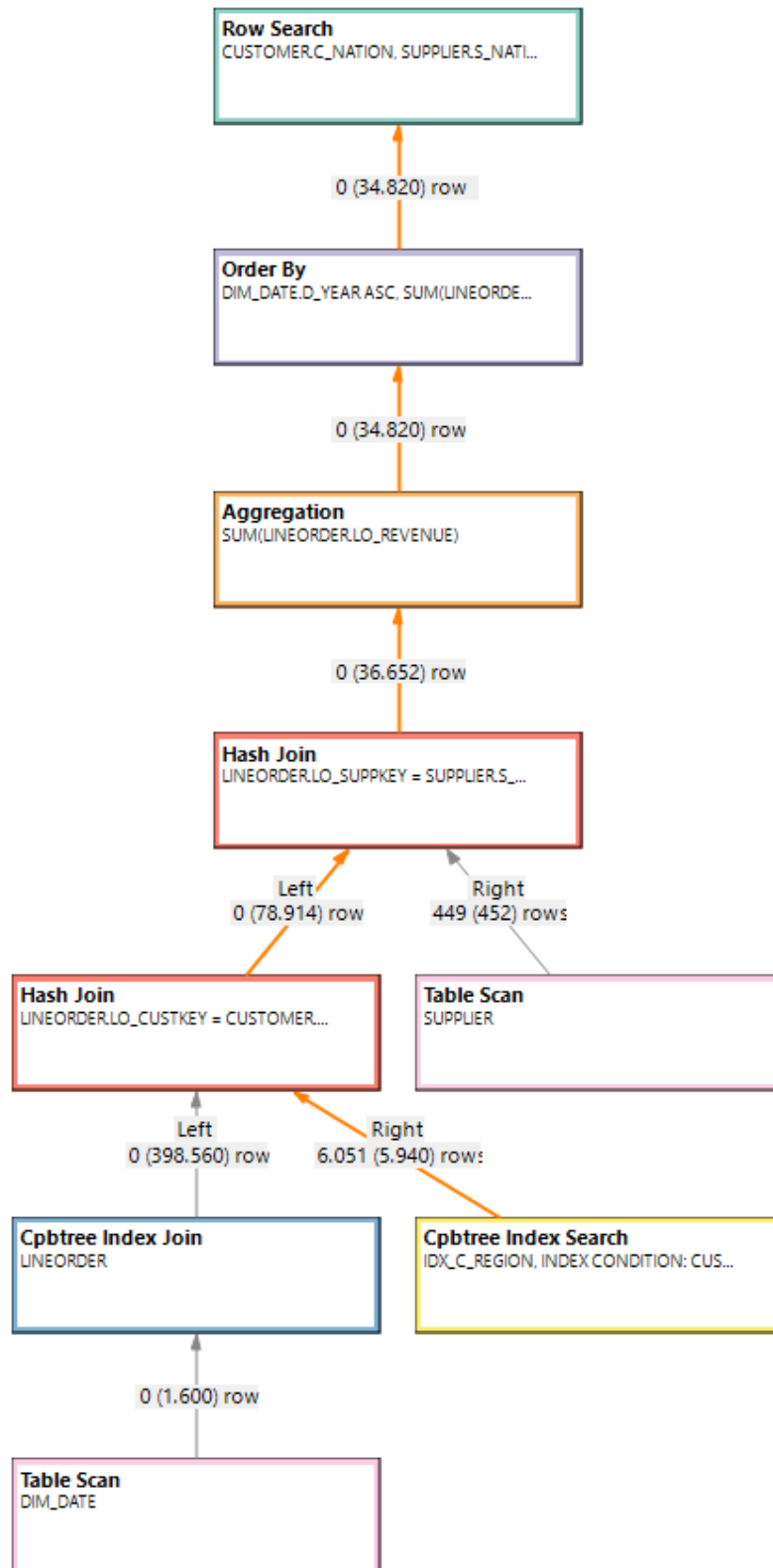


Abbildung A2: Q3.1 Execution Plan - Row Store

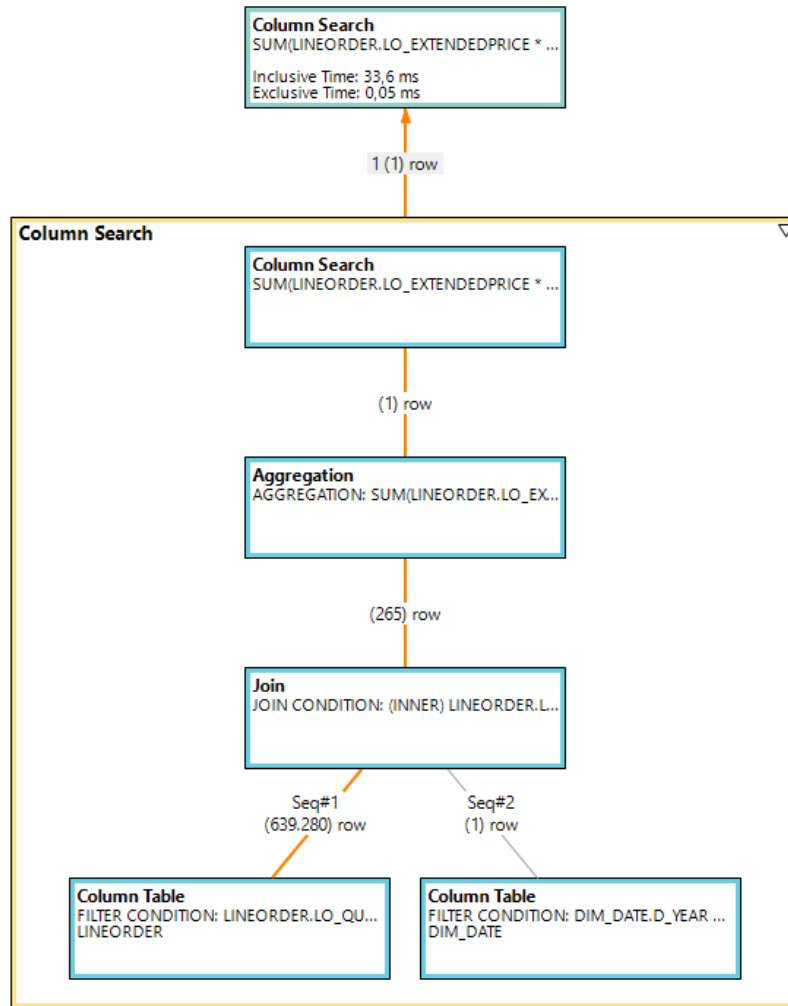


Abbildung A3: Q1.1 Execution Plan - Column Store

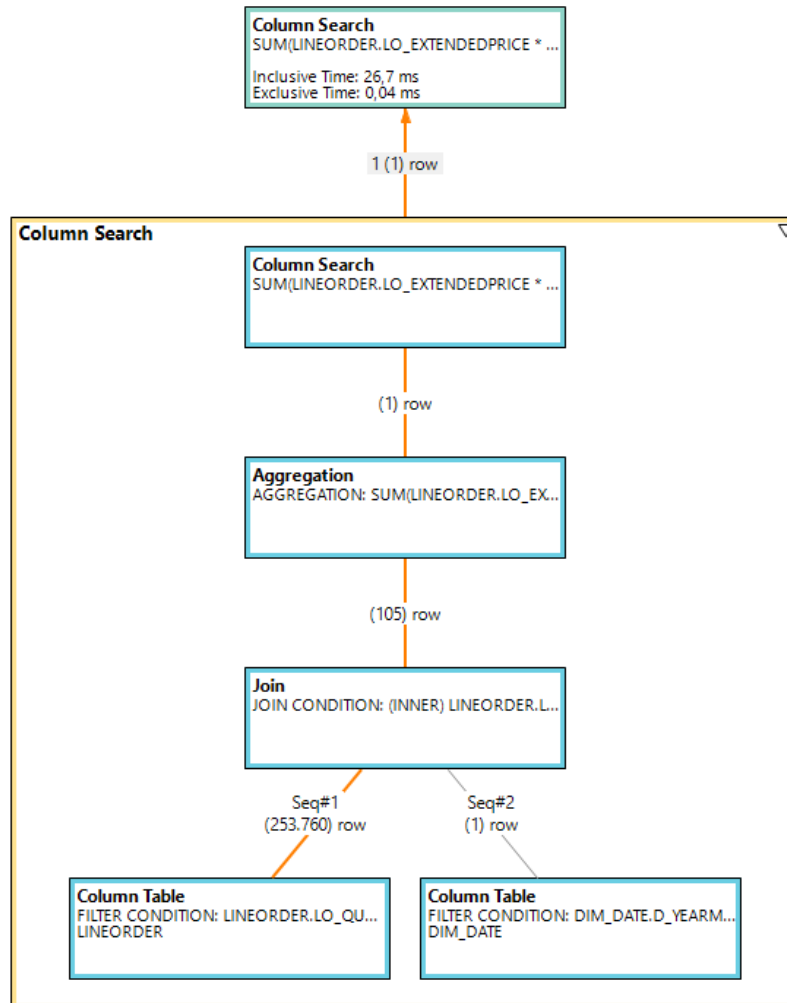


Abbildung A4: Q1.2 Execution Plan - Column Store

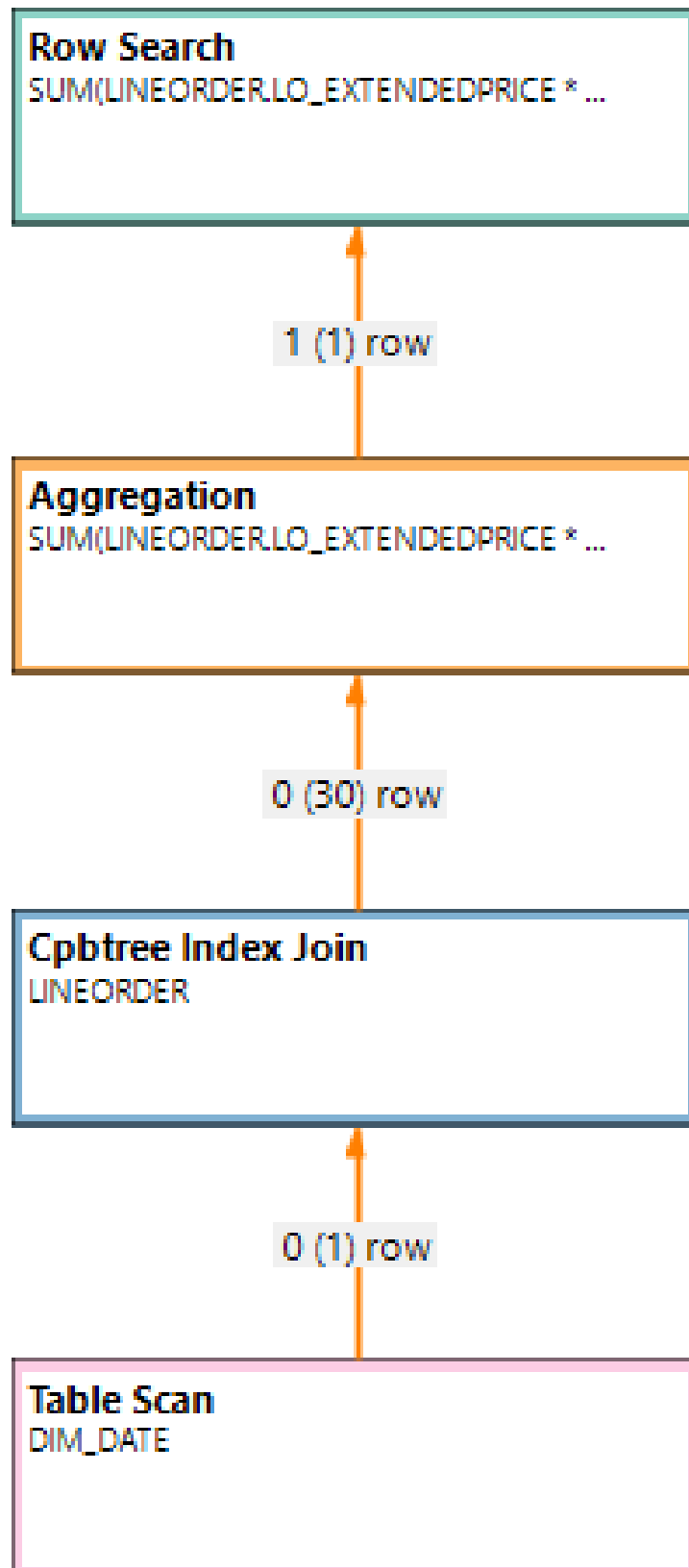


Abbildung A5: Q1.1 Execution Plan - Row Store

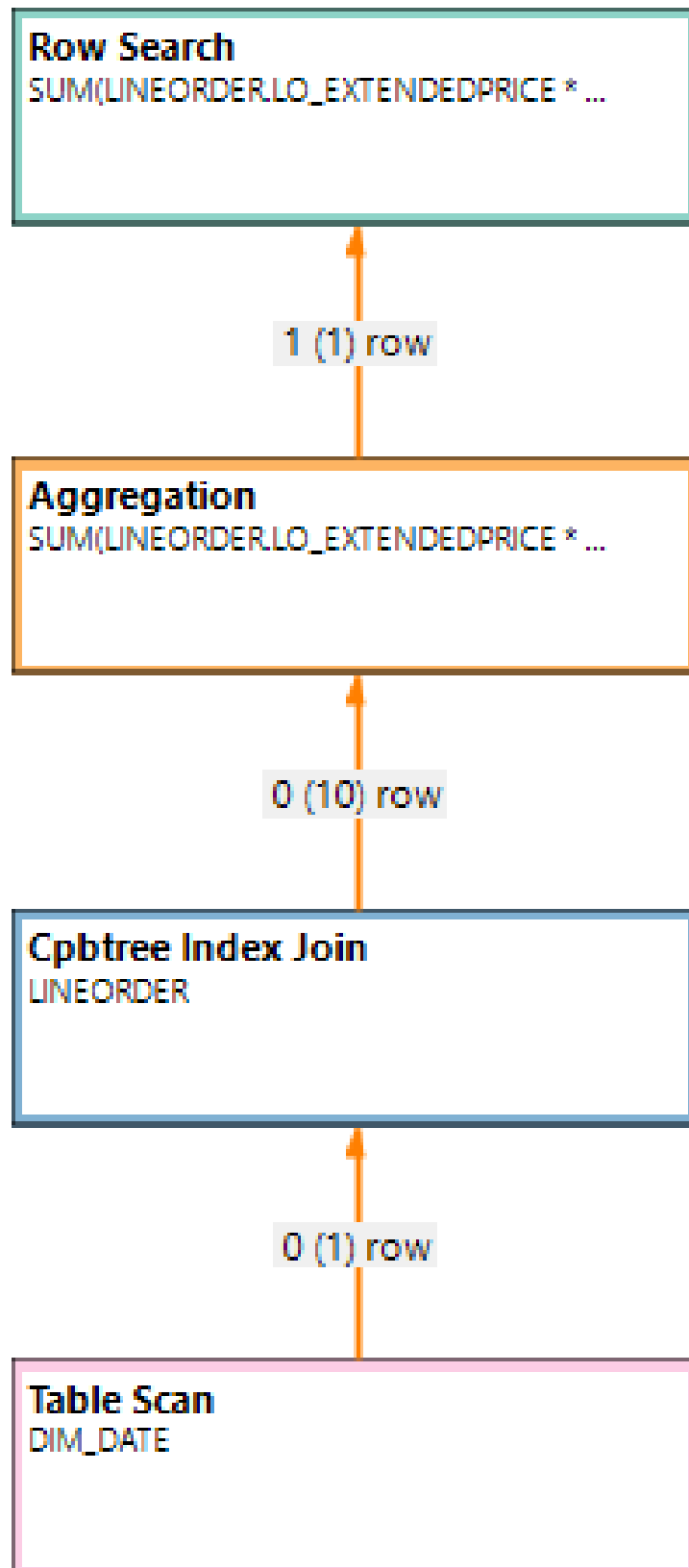


Abbildung A6: Q1.2 Execution Plan - Row Store

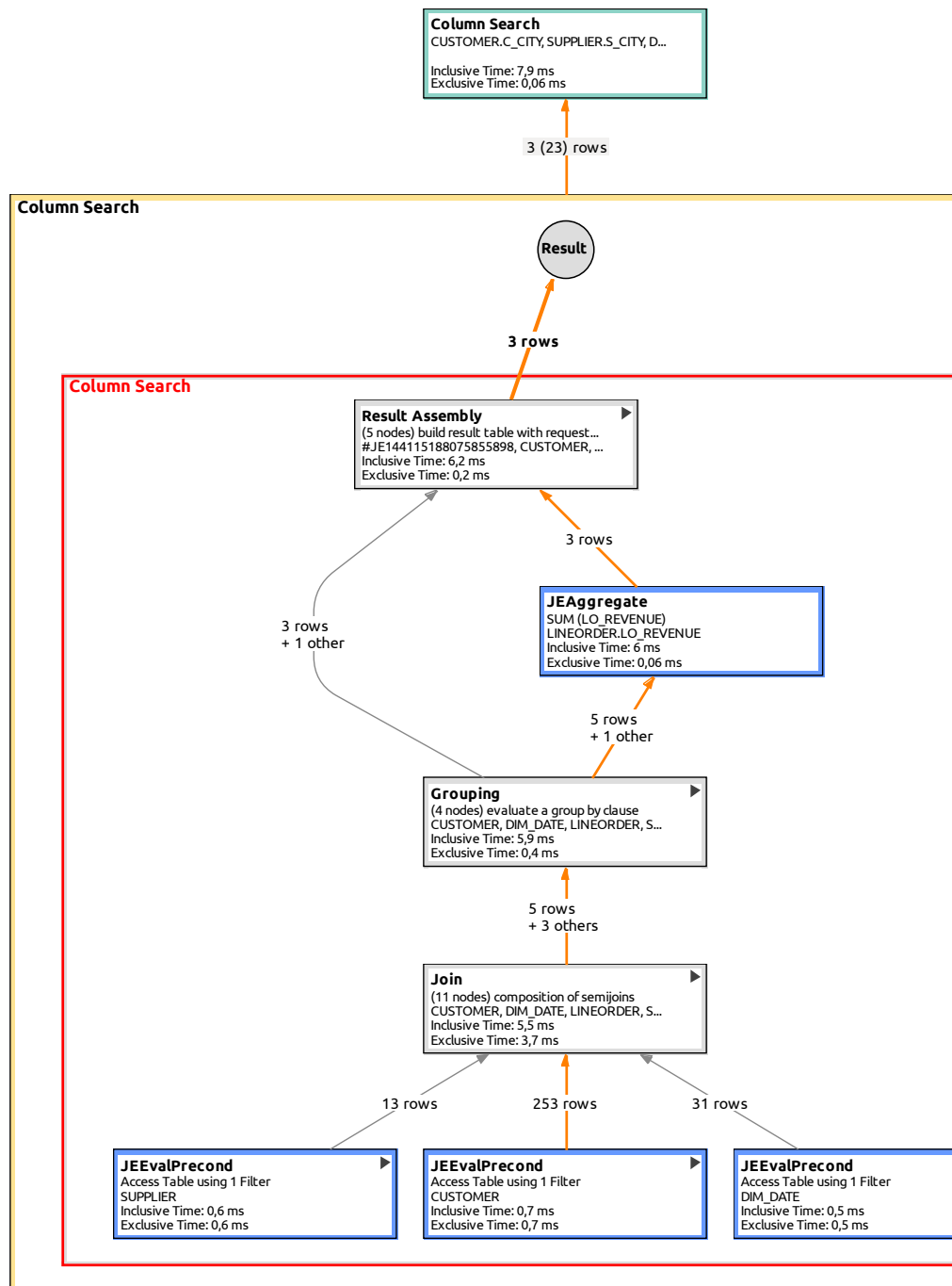


Abbildung A7: Q3.4 Execution Plan - Column Store mit Indizes



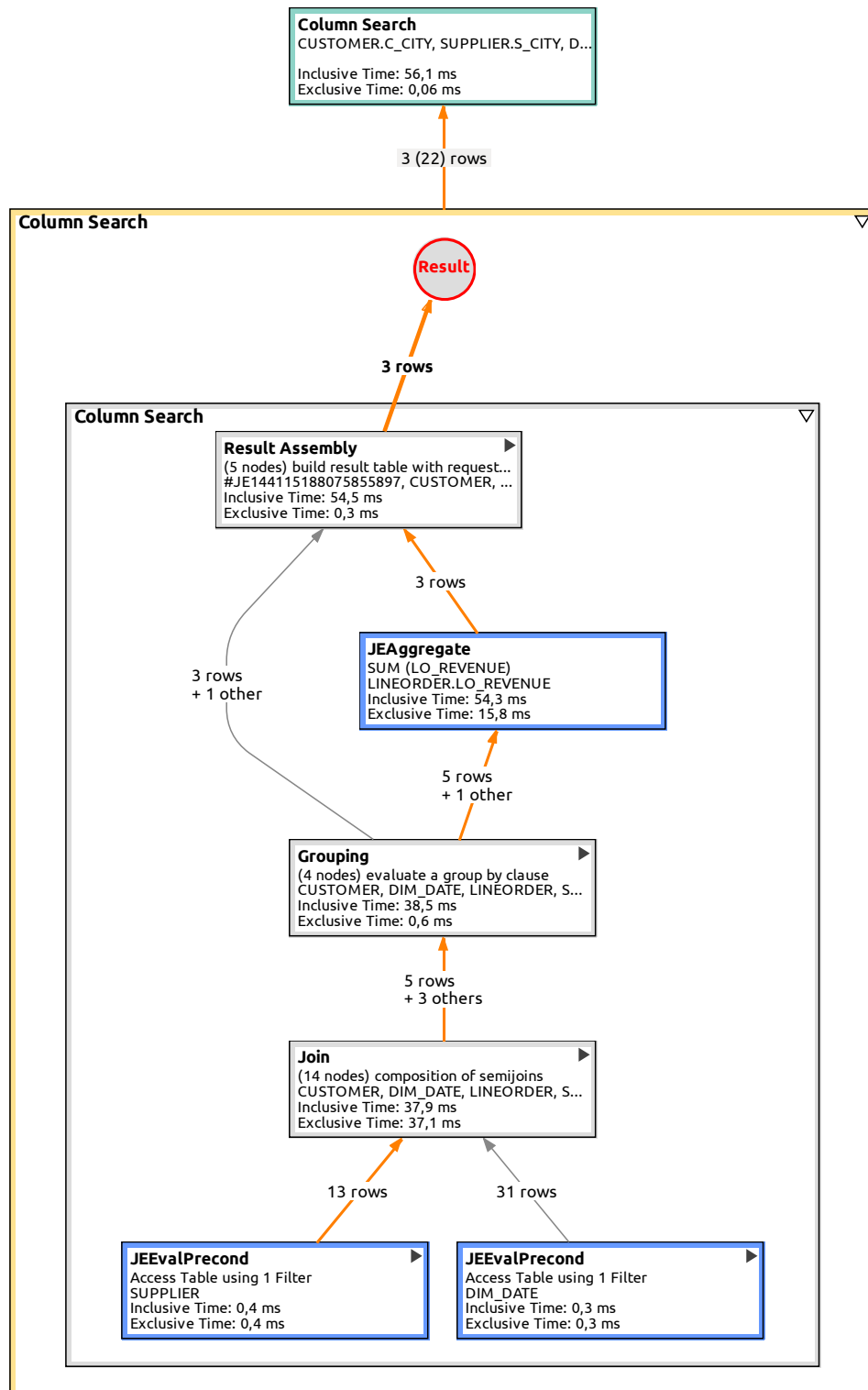
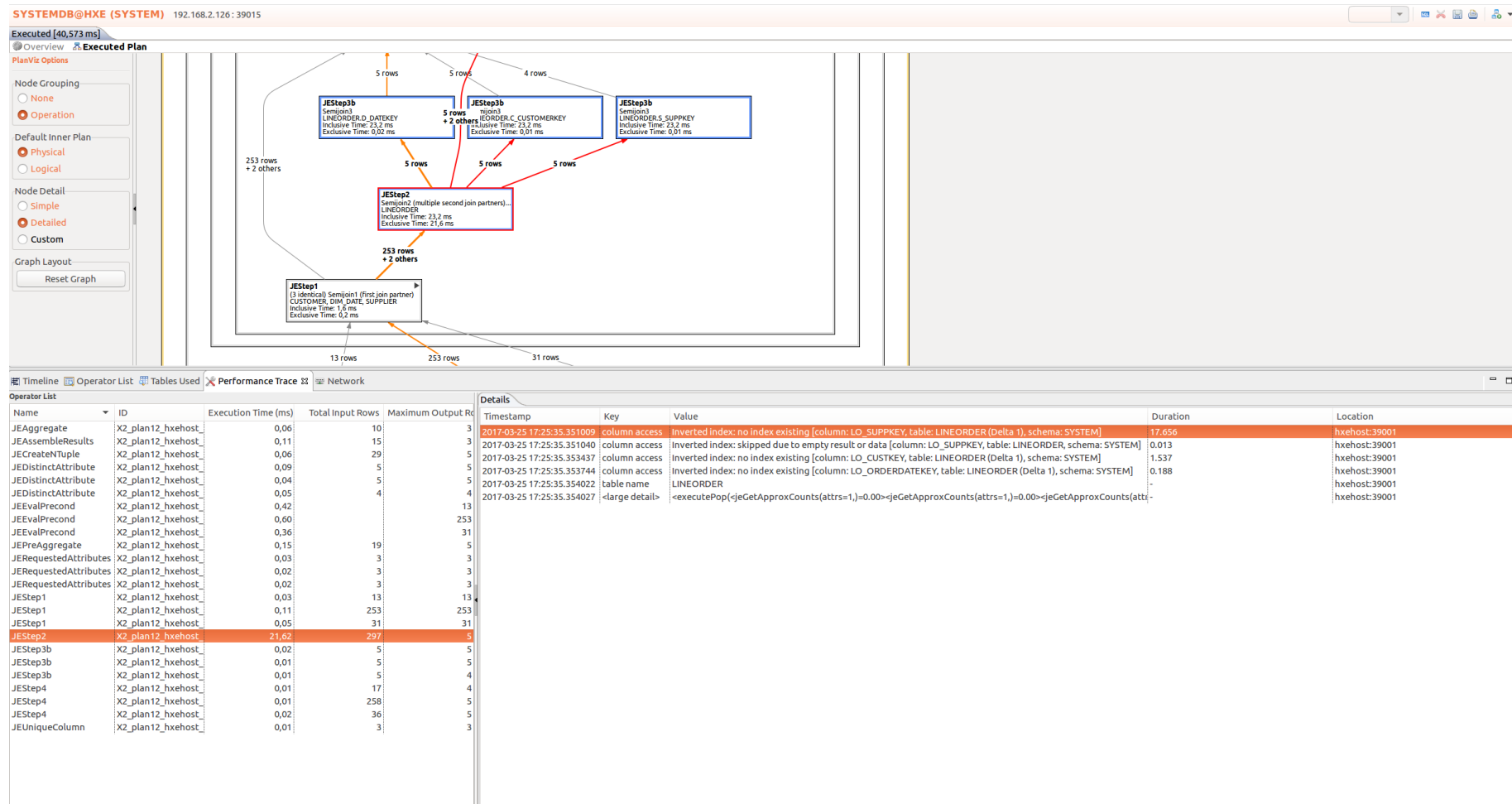


Abbildung A8: Q3.4 Execution Plan - Column Store ohne Indizes



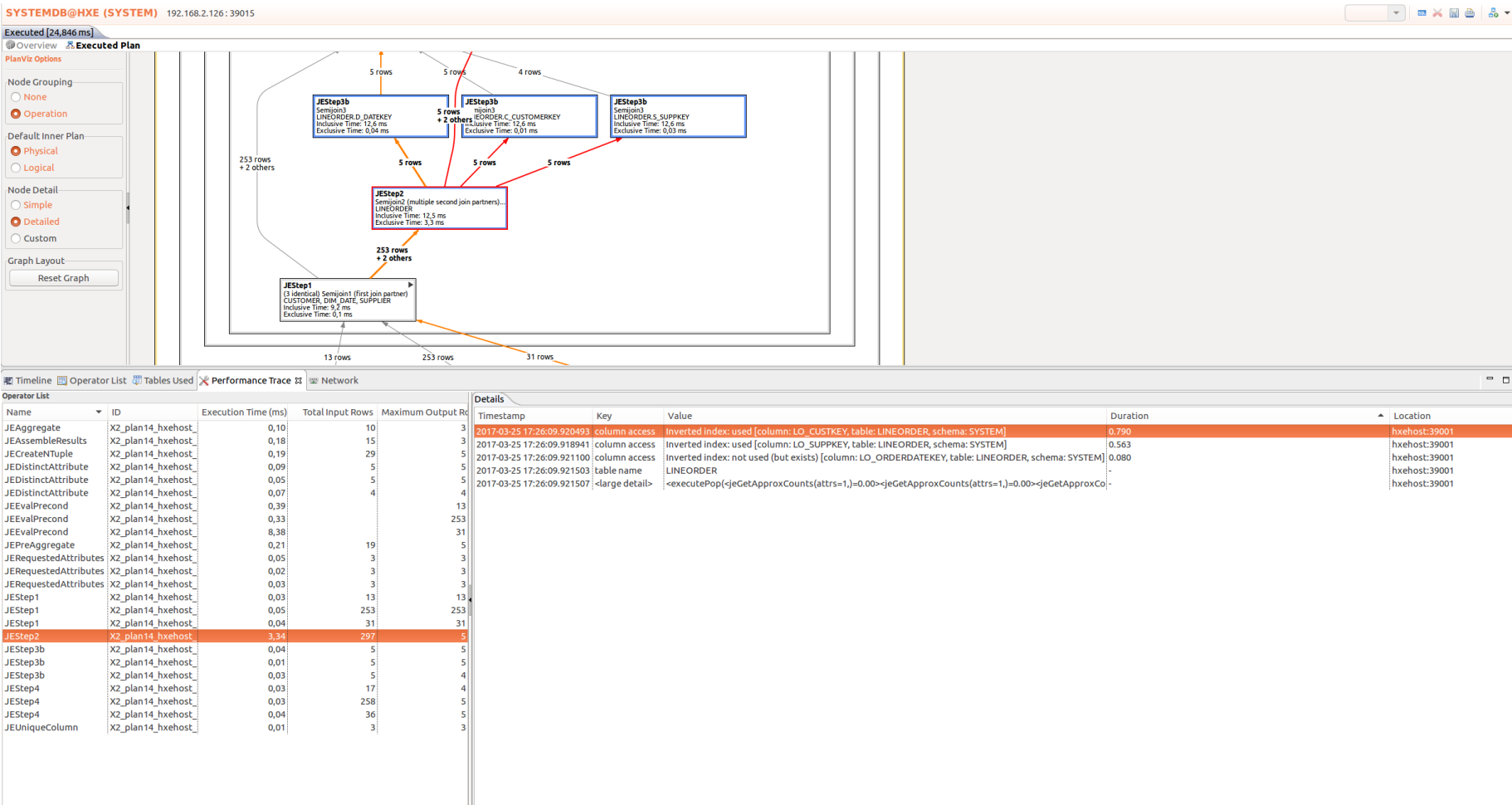


Abbildung A10: Execution Plan: Q3.4 with Index