Der Star Schema Benchmark mit der SAP HANA Datenbank

Seminararbeit

für die Vorlesung

Data Warehouse

des Studiengangs Angewandte Informatik International Business Competence

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Jendrik Jordan, Arwed Mett, Simon Oswald, Dominic Steinhauser

13. Mai 2017

Kurs	TINF14AIBC
Matrikelnummern	1807718, 4278042, 6594512, 7344208
Gutachter	Prof. Dr. Colgen

Erklärung

Wir versichern hiermit, dass wir unsere Seminararbeit 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.

Mannheim, 13. Mai 2017

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. Rowstore, in In-Memory Datenbanktabellen aufgezeigt. Insbesondere werden Laufzeitunterschiede, Verbesserungsmöglichkeiten der Laufzeit mit Indizes, sowie die Auswirkungen von verschiedenen Hardwarevoraussetzungen eines Coulmnstores und Rowstores betrachtet und analysiert. Hierbei hat sich gezeigt, dass der Columnstore in allen Belangen deutlich schneller ist, als der Rowstore. Durch Indizes lässt sich dieser sogar noch beschleunigen, wobei die schnellsten Durchläufe durch Nutzung der OLAP-Engine erzielt werden konnten.

Inhaltsverzeichnis

Abbildungsverzeichnis V

Tabellenverzeichnis VI

Listings VII

1 Einleitung 1
- 1.1 Ziele . . . 1
- 1.2 Aufbau . . . 1

2 Columnstore und Rowstore im Vergleich 2
- 2.1 Der Columnstore . . . 2
- 2.2 Der Rowstore . . . 3
- 2.3 Rowstore und Columnstore im direkten Vergleich im Umfeld der SAP HANA 3

3 Star Schema Benchmark 5
- 3.1 Merkmale . . . 5
- 3.2 Datenbanklayout . . . 6

4 Setup 7
- 4.1 Github Repository . . . 7
- 4.2 Virtuelle Maschine . . . 7
- 4.3 Eclipse . . . 8
- 4.4 SAP HANA HDBSQL . . . 9
- 4.5 Datenbankschema . . . 9
- 4.6 Daten Import . . . 10
- 4.7 Query Execution Plan . . . 11

5 Ausführung des Benchmarks 12
- 5.1 Ziele . . . 12
- 5.2 Realisierung der Ziele . . . 13
- 5.3 Durchführung . . . 14
- 5.4 Verwendung des Benchmarks . . . 14

6 Auswertung Benchmark 17
- 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 Fazit . . . 32

6.6 Auswirkung der OLAP-Engine 33
6.7 Auswirkung unterschiedlicher Hardwarekonfiguration 35

7 Fazit **39**

8 Autoren - Wer hat was geschrieben **40**

Literatur **41**

Anhang **42**

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 - Columnstore 22
6.6 Join - Query Execution Plan für Q4.3 - Columnstore 23
6.7 Query Execution Plan für Q4.3 - Rowstore 24
6.8 Übersicht des DB-Layouts mit Indizes. 25
6.9 Vergleich der Gesamtlaufzeit mit OLAP-Hint. n=100 33
6.10 Benchmark im Columnstore unter Variation der CPU Kerne 35
6.11 Benchmark im Columnstore unter Variation des RAMs 36
6.12 Benchmark im Rowstore unter Variation der CPU Kerne 37
6.13 Benchmark im Rowstore unter Variation des RAMs 38

A1 Q3.1 Execution Plan - Comlumn Store 47
A2 Q3.1 Execution Plan - Rowstore 48
A3 Q1.1 Execution Plan - Columnstore 49
A4 Q1.2 Execution Plan - Columnstore 50
A5 Q1.1 Execution Plan - Rowstore 51
A6 Q1.2 Execution Plan - Rowstore 52
A7 Q3.4 Execution Plan - Columnstore mit Indizes 53
A8 Q3.4 Execution Plan - Columnstore ohne Indizes 54
A9 Execution Plan: Q3.4 55
A10 Execution Plan: Q3.4 with Index 56
A11 QEP für Query 2.3 ohne Index. 58
A12 QEP für Query 2.3 mit Index. 59
A13 QEP für Query 4.1 ohne Index. 60
A14 QEP für Query 4.1 mit Index. 61

Tabellenverzeichnis

4.1 Dauer des Importieren der Daten in Column- und Rowstore 10

6.1 Gesamtlaufzeiten 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 Columnstore. 26
6.7 Durchschnittslaufzeit für jede Benchmarkgruppe für Columnstore. 27
6.9 Durchschnittslaufzeit für Benchmarkgruppen 1 und 3 für Columnstore. . . 27
6.10 Durchschnittslaufzeit für Query 1.2 bei Columnstore. 28
6.12 Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Rowstore. 30
6.14 Durchschnittslaufzeit jeder Querygruppe. n=250 30
6.16 Laufzeiten der Queries aus Gruppe 2. 31
6.18 Laufzeiten der Queries aus Gruppe 4. 31
6.19 Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Columnstore. 34
6.21 Laufzeit jeder Benchmarkgruppe für Columnstore mit Index, testweise mit OLAP-Hint. n=100 34
6.22 Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Rowstore. 34

Listings

4.1 Ordnerstruktur 8
4.2 HDBSQL Befehl 9
4.3 Beispiel SQL-Befehl: Datenimport 10

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 Columnstore 42
2 Schema Rowstore 44
3 Indizes hinzufügen 46
4 import.sql 46

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 (Rowstore vs. Columnstore) 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 Rowstore
2. Vergleich der Ausführungszeiten des Benchmarks mit bzw. ohne Indizes
3. Vergleich der Ausführungszeiten des Benchmarks bei verschiedenen Hardwarevoraussetzungen

1.2 Aufbau

In Kapitel 2 werden die grundlegenden Unterschiede von Column- und Rowstore 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. Anschließend werden die Ergebnisse in Kapitel 6 untersucht und analysiert. Kapitel 7 beinhaltet eine Zusammenfassung der Ergebnisse der Analyse.

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 [vgl. 4, 13]:

- **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**: Zugriffe, 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. Wurden Einträge vom Wert X zu einer Einheit Y komprimiert, kann bei einer Suche nach allen Einträgen mit Wert X direkt Einheit Y zurückgegeben werden. Genauso könnte bei einer Aggregatfunktion, wie dem Errechnen des Durchschnittes aller Werte, alleinig auf den Wert X von Einheit Y zurückgegriffen werden, ohne jeden Eintrag einzeln abfragen zu müssen.
- **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 [vgl. 1, 5]:

- **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 Datensätze 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. Auschlaggebend für die Entscheidung zu Row- oder Columnstore ist aber vorallem 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 allerdigs

die Notwendigkeit stets den kompletten Datensatz lesen zu müssen, obwohl möglicherweise nur bestimmte Spalten benötigt werden.

Sind Anfragen vorallem 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. Einfügeoperationen, welche Werte einer nicht vorgesehenen Größe beinhalten, ziehen eine Neu-Organisation der Daten mit sich, wodurch die Integrität der Indexierung der Datensätze sichergestellt wird.

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 [vgl. 16].

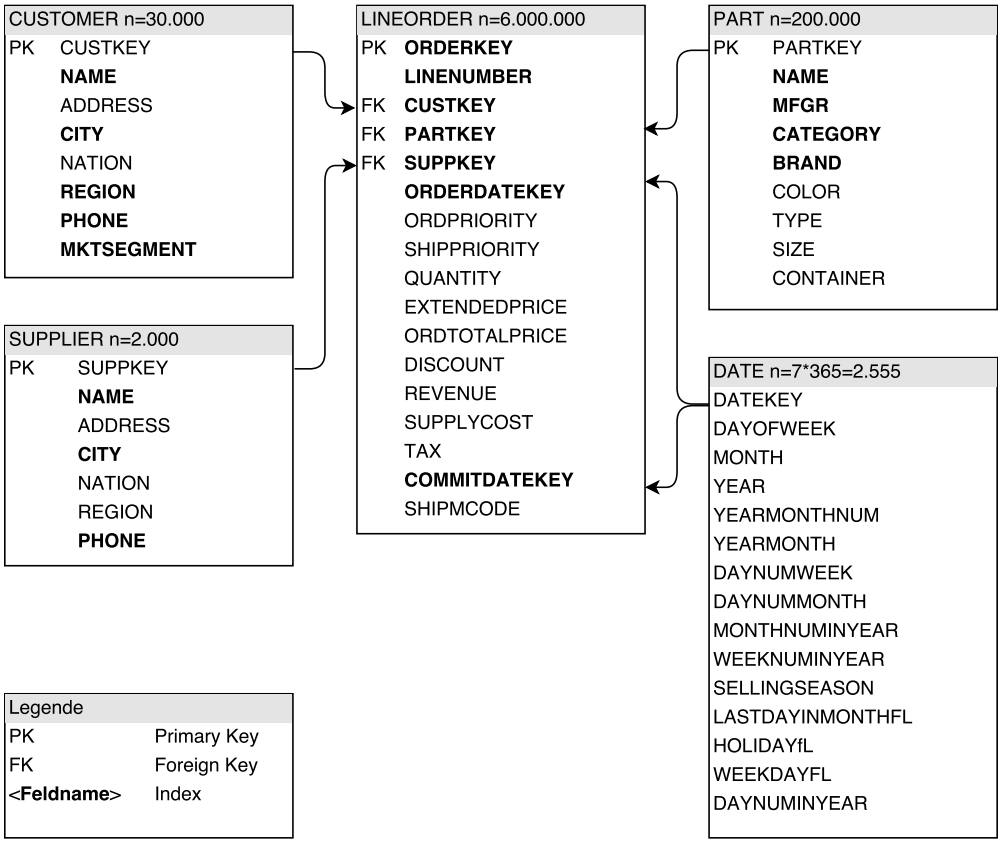
Das Ergebnis des TPC-H Benchmark ist die „TPC-H Composite Query-per-Hour“ -Metrik. Durch sie werden Aspekte wie die genutzte Datenmenge oder das parallele Anfragestellen mehrerer Nutzer berücksichtigt. Damit soll sichergestellt werden, dass nur valide Vergleiche von Ergebnissen stattfinden können, da bei unterschiedlichen Vorraussetzungen irreführende Ergebnisse aus dem Vergleich resultieren können [vgl. 16].

3.1 Merkmale

Die Merkmale des Benchmarks sind wie folgt [vgl. 15]:

- **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 gleichzeiten Ändern des Datenbestandes, sowie dem Lösen von kritischen Entscheidungen.
- **Constraints**: Eine große Menge an unterschiedlichen Constraints werden eingesetzt sowie vorrausgesetzt für das operierende System, wodurch eine realitätsnahe Bedingungen geschaffen werden.
- **Relevanz**: Das Schema und die Anfragen orientieren sich an relaventen real-Anwendungsfällen.
- **Datenmenge**: Die Minimalanforderung an die Datenmenge sieht 2 Tausend Datensätze in der „SUPPLIER“ -Tabelle vor, wodurch ca. 1 Gigabyte an Daten vorliegen.

3.2 Datenbanklayout



4 Setup

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

4.1 Github Repository

Alle für die Arbeit notwendigen Dokumente bzw. Skripte sind unter folgendem Link[1] in Github verfügbar. Im „src“-Verzeichnis liegen die Skripte zum Ausführen des Benchmarks. Das Verzeichnis „src/sql“ enthält dabei alle notwendigen SQL-Dateien, die beispielweise für das Erstellen von Tabellen, Importieren der Daten oder Ausführen von Abfragen, genutzt werden.

4.2 Virtuelle Maschine

Unter dem folgenden Link[2] kann eine SAP HANA Instanz als virtuelle Maschine heruntergeladen werden. Für den initialen Setup, Benutzername und anlegen eines Passworts, ist dieser Link[3] 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 bash-Skripte, sowie .sql-Dateien aus Github[4] im Verzeichnis „/usr/sap/HXE/HDB90/work/“ der Virtuellen Maschine abgelegt sind (vgl. Listing 4.1).
Die Daten für den SSBM lassen sich über folgenden Link[5] eigenständig generieren. Die

1 https://github.com/Osslack/HANA_SSBM
2 https://www.sap.com/developer/topics/sap-hana-express.html
3 https://www.sap.com/developer/tutorials/hxe-ua-getting-started-vm.html
4 https://github.com/Osslack/HANA_SSBM/tree/master/src
5 https://github.com/electrum/ssb-dbgen

erzeugten .tbl-Dateien müssen dann ebenfalls in der Virtuellen Maschine im „work“-Verzeichnis abgelegt werden (vgl. Listing 4.1).

```
-work
  -all_benchmarks.bash
  -hdbsql.bash
  -run_benchmark.bash
  -customer.tbl
  -date.tbl
  -lineorder.tbl
  -part.tbl
  -supplier.tbl
  -sql
    -addBasicIndizes.sql
    -advancedIndizes.sql
    -bench_all.sql
    -furtherIndizes.sql
    -import.sql
    -schemaCol.sql
    -schemaRow.sql
    -benchmark
```

Listing 4.1: Ordnerstruktur

Zum Starten des Benchmarks muss lediglich die Datei „run_Benchmark.bash“ gestartet werden.

4.3 Eclipse

Eclipse wird in der Regel als Entwicklungsumgebung für die SAP HANA Datenbank verwendet. Außerdem lassen sich in Eclipse Query Exection Pläne grafisch darstellen.
Unter dem folgenden Link[1] ist beschrieben, welche Erweiterungen und Einstellungen in „Eclipse“ vorgenommen werden müssen, um eine Verbindung zur Datenbank herzustellen.
Zum einfachen Ausführen unseres Benchmarks und Vergleichen der Laufzeitunterschiede reichen die zur Verfügung stehenden Mittel von HDBSQL vollkommen aus.
Möchten man sich allerdings Query Execution Pläne grafisch darstellen, so wird Eclipse benötigt.

[1] https://www.sap.com/developer/how-tos/2016/09/hxe-howto-eclipse.html

4.4 SAP HANA HDBSQL

SAP HANA HDBSQL ermöglicht das Ausführen von SQL Befehlen in der Kommandozeile[1]. 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

4.4.1 Übersicht Parameter HDBSQL

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

4.5 Datenbankschema

Zu Beginn muss das Datenbankschema des SSBM-Benchmarks definiert werden. Die Tabellen werden entweder als Columnstore (vgl. Listing 1) oder in einem Rowstore 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 Columnstore zuerst ohne Indizes ausgeführt. Diese lassen sich anschließend über das Statement „CREATE Index <index_name> ON <tabellen_name>“ hinzufügen (vgl. Listing 3).

[1] http://help-legacy.sap.com/saphelp_hanaplatform/helpdata/en/c2/2c67c3bb571014afebeb4a76c3d95d/content.htm?frameset=/en/c2/4d054bbb571014b253ac5d6943b5bd/frameset.htm&current_toc=/en/00/0ca1e3486640ef8b884cdf1a050fbb/plain.htm&node_id=1155&show_children=false

4.6 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 angelegt wurde, können nun die SSBM-Benchmark-Daten importiert werden. Das Importieren der Daten wird in Listing 4 dargestellt. In diesem Fall liegen die Daten als .tbl-Dateien vor. Die Dateiendung spielt allerdings keine Rolle, solange die Struktur der Daten einer CSV-Datei entspricht.
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

Tabellenname	Columnstore	Rowstore
DIM_Date	84 ms	109 ms
Customer	357 ms	940 ms
Lineorder	1:13:233 min	2:56:135 min
Part	1,89 s	6,122 s
Supplier	118 ms	104 ms
Gesamt	1:15:685 min	3:03:401 min

Tabelle 4.1: Dauer des Importieren der Daten in Column- und Rowstore[1]

[1] Hardwarespezifikation der VM: RAM: 6GB, Prozessor:Intel Core-i7-860, Kerne: 4

Beim Importieren der Daten in die Datenbank fällt auf, dass bei beiden Speicherarten (Column- & Rowstore) die Tabelle Lineorder am längsten benötigt. Dies ist auf die große Anzahl der Datensätze zurückzuführen.
Das Importieren in den Rowstore dauert bei der Tabelle Lineorder ca. 2,41 mal solange wie in den Columnstore, bei der Tabelle Customer wird ca. 2,63 mal so viel Zeit benötigt und bei der Tabelle Part ist der Unterschied der Laufzeiten am größten zwischen spalten- bzw. zeilenbasierter Speicherung der Daten, nämlich um den Faktor 3,24.

4.7 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 **erleichtete 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 zwischenzuspeichern. Zu diesem Zwecke werden während der Durchführung der Benchmarks **Logs** angelegt, welche die Daten halten.

Ein weiterer Aspekt ist die **Anzahl an Iterationen** innherhalb 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 angesprochende inhaltliche Variation des Benchmarks realisieren.

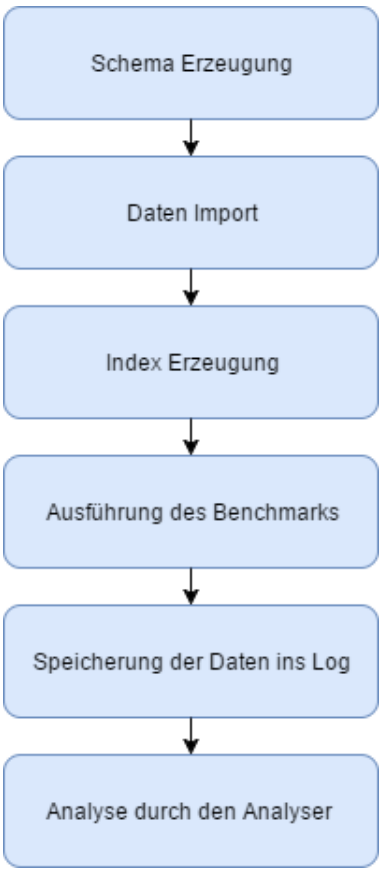


Abbildung 5.1: Durchführung

Nach erfolgreicher Anmeldung über die Zugangsdaten zur HANA Instanz erfolgt zuerst der Benchmark auf Basis eines Columnstores ohne Indizies. 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 Gigybyte RAM.

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[3] 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 gefunden werden.

Durch ausführen des run_benchmark.sh scripts kann der Benchmark gestarted 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. Möchte man nicht die Standardeinstellungen verwenden, muss das Script mittels `run_benchmark.sh -v` ausgeführt werden. Dadurch bekommt man bei jedem Schritt die Möglichkeit, die Einstellung anzupassen.

Um den Benchmark anzupassen können neue Tests dem Script hinzugefügt werden. Die Idee dabei ist, dass der Tester dem Framework sagt, wie seine Benchmarks strukturiert sind und anschließend mittels SQL[8] und BASH spezifische Queries ausführen und wiederholen kann. Anschließend werden die Laufzeitdaten der SQL Queries als JSON[10] in einer Log Datei gespeichert. Dann kann ein Analyst mit seiner favorisierten Programmiersprache die Daten auswerten. In unserem Fall erfolgte die Auswertung in Python unter zu Hilfe nahme von Jupyter Notebook zur Visualisierung der Ergebnisse.

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 gestarted 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 ließt 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.

```
source ./hdbsql.bash
source ./all_benchmarks.bash

hdb_init_log

hdb_start_benchmark <benchmark name>
run_all_benchmarks <path> <repetitions>
hdb_end_benchmark

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.

```
{
  "column_benchmark_no_index": [
    [
      {
        "Type": "exec_file",
        "Filename": "./sql/benchmark/q1_bench/q1.sql",
        "times": " 88645;70697;39871;"
      },
      {
        "Type": "exec_file",
        "Filename": "./sql/benchmark/q1_bench/q1.1.sql",
        "times": " 44470;"
      }
    ]
  ]
}
```

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 spatenbasierten 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. Anschließen werden in Abschnitt 6.3 beispielhaft die Query Execution Pläne zwischen Column- und Rowstore verglichen. In Abschnitt 6.4 und Abschnitt 6.7 wird der Einfluss von Indizes bzw. unterschiedlicher Hardwarekonfigurationen untersucht. In Abschnitt 6.6 wird die Ausuwirkung der OLAP-Engine genauer analysiert.

6.1 Gesamtlaufzeit des Benchmarks

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.

	Columnstore	**Rowstore**
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

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 Data-

warehouse 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.

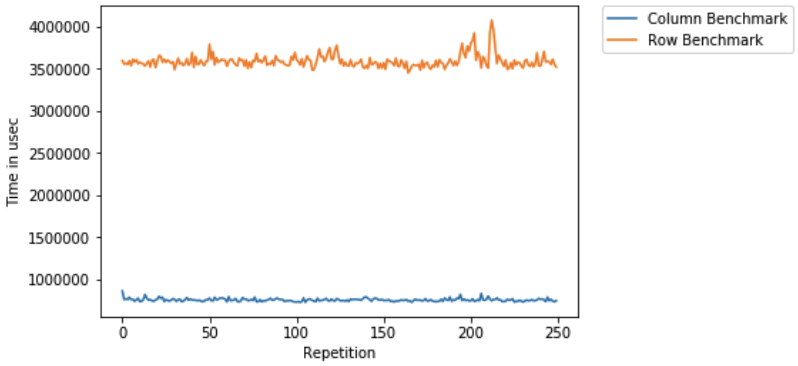


Abbildung 6.1: Gesamtlaufzeit von Row- und Columnstore

Anhand der Standardabweichung und Abbildung 6.1 ist auch zu sehen, dass der Columnstore eine konstantere Zeit pro Abfrage aufweist. Da der Rowstore allerdings 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%.

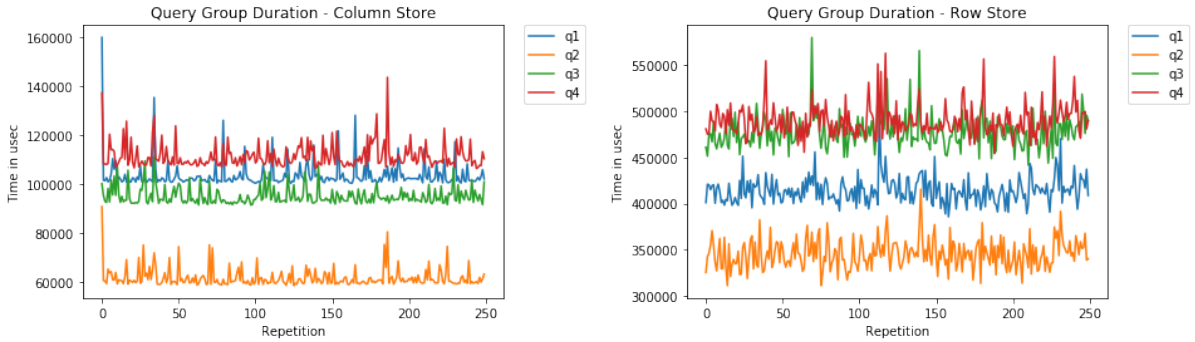


Abbildung 6.2: Query-Gruppen Laufzeit

In Abbildung 6.2 ist zu erkennen, dass die Laufzeit einzelner Queries durch zeilenbasiete 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 Rowstore betrachtet. Anschließend werden die selben Queries im Rowstore 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 Unterabfagen von Q3 betrachtet. Ziel ist es die entscheidenden Queries zu identifizieren, welche durch den Columnstore bzw. den Rowstore bevorzugt werden.

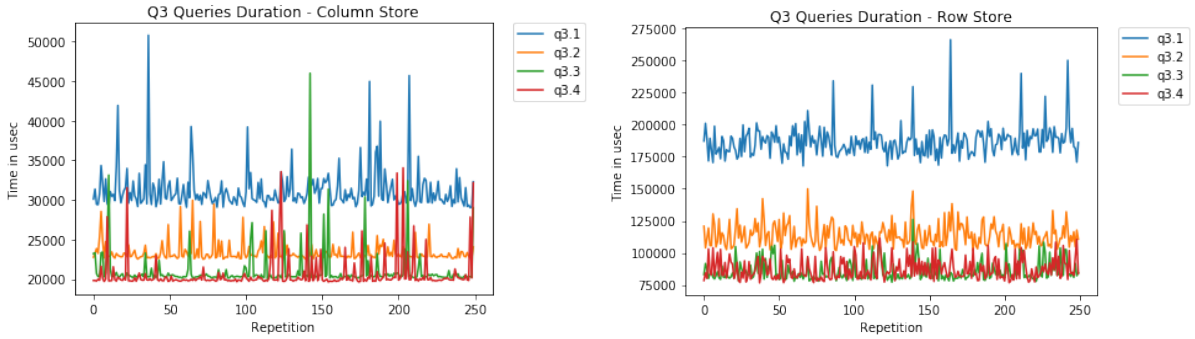


Abbildung 6.3: Q3 Vergleich Row vs Column

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.

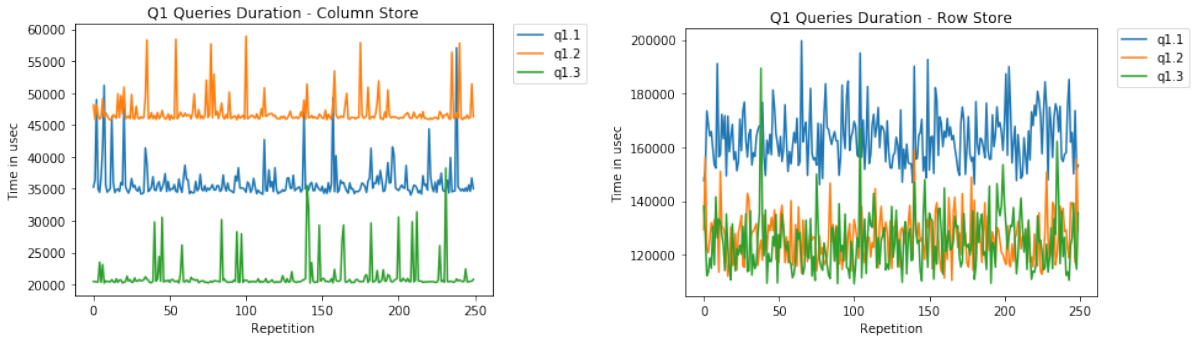


Abbildung 6.4: Q1 Vergleich Row vs Column

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

Um der Ursache für den großen Laufzeitunterschied zwischen Row- und Columnstore auf den Grund zu gehen, werden im folgenden exemplarisch die Query Execution Pläne für Query 4.3 untersucht, siehe Listing 6.1. Die Wahl viel hierbei auf Q4.3, da er einerseits deutlich beschleunigt wurde, siehe Tabelle 6.3, andererseits handelt es sich um einen der komplexeren Queries, wovon wir uns interessantere Ergebnisse erhoffen.

```
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 - Columnstore

Abbildung 6.5 zeigt den Ablauf der Query Execution im Columnstore. Es ist zu erkennen, dass zu Beginn jede Tabelle einzeln nach den drei angegebenen Kriterien gefiltert wird.

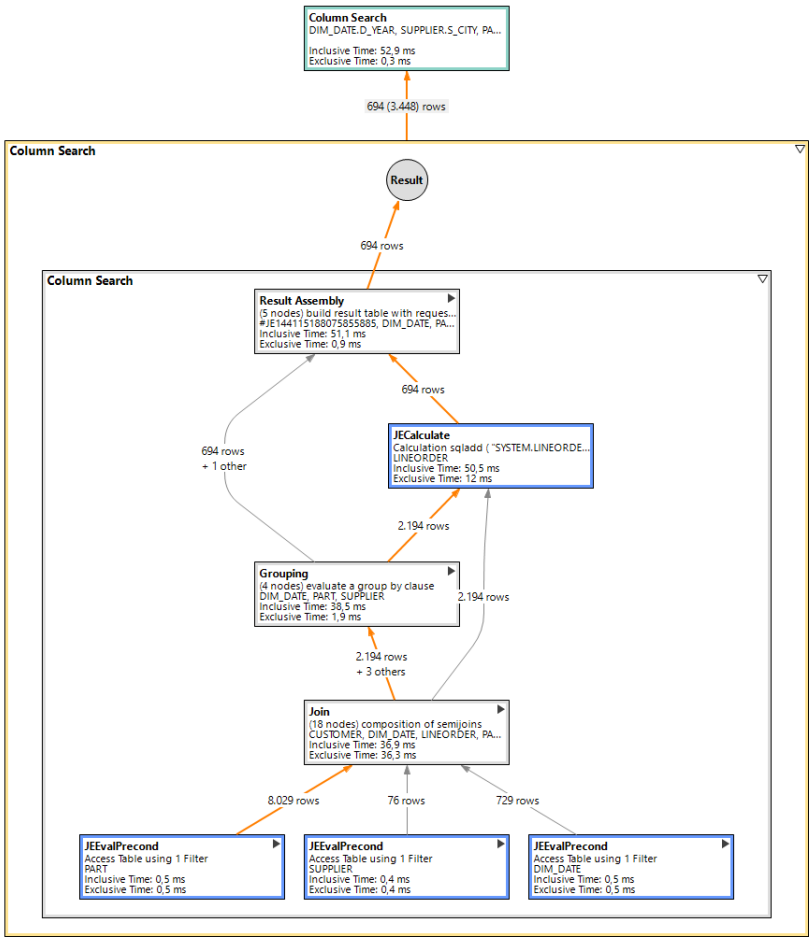


Abbildung 6.5: Query Execution Plan für Q4.3 - Columnstore

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 **Lineorder** gejoint.

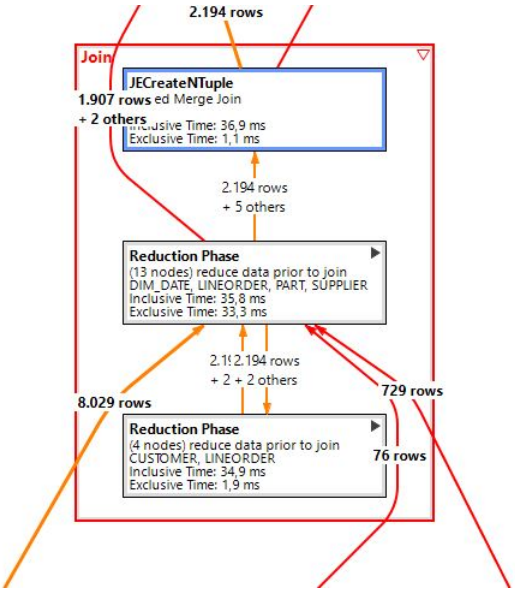


Abbildung 6.6: Join - Query Execution Plan für Q4.3 - Columnstore

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 ausgeben.

6.3.2 Query Execution Plan - Rowstore

Abbildung 6.7 zeigt den Query Execution Plan für das gleiche Query, nur dass die Daten im Rowstore abgelegt sind. Ohne genaueres Hinschauen fällt auf, dass die Ausführung sich im Vergleich zum Columnstore stärker an einem sequentiellen Ablauf orientiert.
Im Rowstore 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 **Lineorder.LO_Partkey** und **Part.P_Partkey** ausgeführt. (9118 Ergebnisse)

5. Zeitgleich zu dem Hash Join kann ein Table Scan auf die Tabelle **DIM_Date** erstellet 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 **Lineorder** 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.

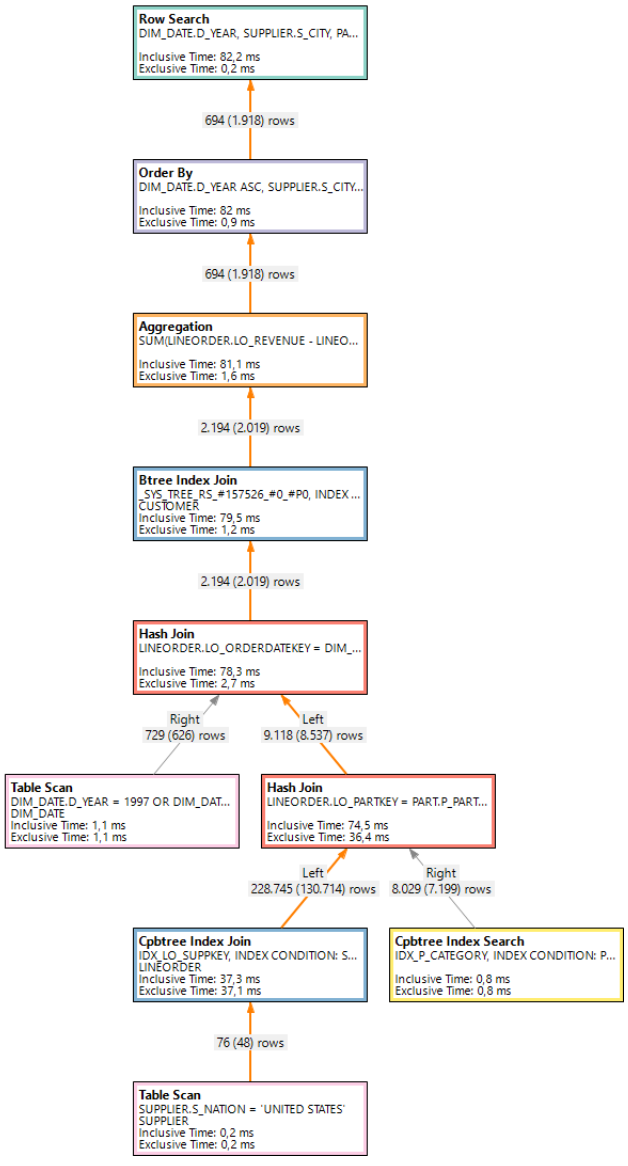


Abbildung 6.7: Query Execution Plan für Q4.3 - Rowstore

6.3.3 Fazit Query Execution Plan

Der große zeitliche Unterschied zwischen Row- und Columnstore lässt sich mit der sequentielle Ausführung des Query im Rowstore begründen.

In beiden Ausführungen war ersichtlich, dass die Ausführung in die 3 Schritte, Filtern, Join und Gruppieren eingeordnet werden kann. Allerdings sind im Rowstore das Filtern und Joinen eng miteinander verbunden und werden sogar abwechselnd nacheinander ausgeführt. Beim Columnstore 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

Im folgenden soll die Auswirkung grundlegender Indizes auf die Laufzeit der Queries untersucht werden. Die angelegten Indizes finden sich in Abbildung 6.8 zeigt das DB-Layout mit Indizes. Die **fett** markierten Attribute sind Felder, auf die Indizes angelegt wurde.

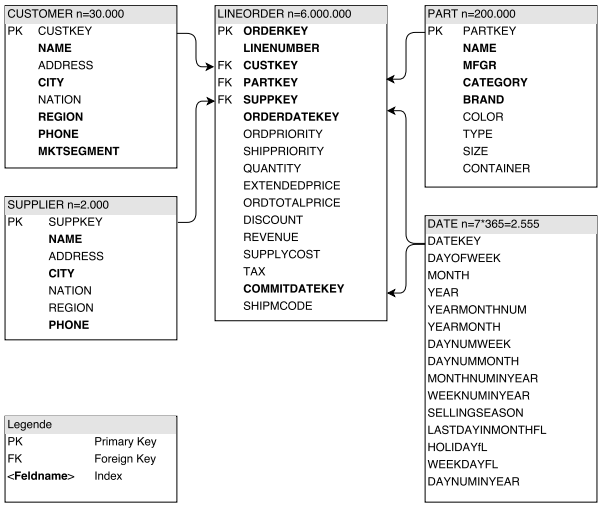


Abbildung 6.8: Übersicht des DB-Layouts mit Indizes.

Um die Indizes anzulegen wurde der folgende Code ausgeführt:

```
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_linenumber 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);
```

6.4.1 Grundlegende Untersuchung für Columnstore

Merkmal	Col[ms]	Col Index[ms]	Abweichung[%]
Samples	250	250	
Median	373	286	23.3%
Average	376	289	23.1%

Tabelle 6.5: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Columnstore.

Durch hinzufügen der grundlegenden Indizes wurde der Benchmark für Columstores sowohl im Schnitt als auch im Median schneller. Im Schnitt wurde er 23,1%, im Median um 23,3% schneller.

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

Schaut man sich die Ergebnisse für jede Query Gruppe (siehe Tabelle 6.7) an, so sind deutliche Verbesserungungen 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.

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 Columnstore.

Hier sind besonders bei Query 1.2 und 3.4 interessant, da diese die größte Verbesserung in ihrer Gruppe vorweisen können, siehe Tabelle 6.9. Die Laufzeit wurde um 64% für Query 1.2 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 1.2 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 Columnstore.

Analyse von 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 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.

Col [ms]	Col Index [ms]
16	6

Tabelle 6.10: Durchschnittslaufzeit für Query 1.2 bei Columnstore.

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.[1]

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

Listing 6.2: Ohne Index

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

Listing 6.3: Mit Index

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

Analyse von 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 Columnstores scheinen sinnvoll angelegte Indizes also einen deutlichen Unterschied zu machen.

Fazit für Columnstores

Auch Columnstores können durch geschickt gewählte Indizes deutlich beschleunigt werden. Erwartet man nur wenige schreibende Zugriffe auf eine Tabelle kann es also durchaus Sinn machen, sinnvolle Indizes anzulegen.

6.4.2 Grundlegende Untersuchung für Rowstore

Durch hinzufügen der grundlegenden Indizes wurde der Benchmark für Rowstores sowohl im Schnitt als auch im Median schneller. Im Schnitt wurde er 15,9%, im Median um 15,9% schneller.

Die Verbesserungen beim Rowstore fallen, zumindest relativ gesehen, deutlich geringer als beim Columnstore (23,1% und 23,3%) aus.

Wert	Row[ms]	Row Index[ms]	Abweichung [%]
Samples	250	250	—
Median	1794	1508	15.9%
Average	1800	1513	15.9%

Tabelle 6.12: Vergleich der Ergebnisse mit und ohne grundlegende Indizes für Rowstore.

Schaut man sich die Veränderung der Laufzeit für die einzelnen Querygruppen, so fällt auf, dass Gruppe 1 und 2 deutlich von den Indizes profitieren, Gruppe 3 so gut wie gar nicht und Gruppe 4 wurde sogar **deutlich** langsamer. Um zu analysieren, woran diese deutlichen Verbesserungen/Verschlechterungen liegen, wird aus Gruppe 2 und 4 jeweils der Query mit der größten Änderung analysiert.

	q1	q2	q3	q4
Row[ms]	423	352	519	507
Row Index[ms]	170	90	473	780
Verbesserung[ms]	253	120	46	-273
Verbesserung[%]	59,8%	74,4%	8,8%	-53,8%

Tabelle 6.14: Durchschnittslaufzeit jeder Querygruppe. n=250

6.4.3 Analyse der Queries aus Gruppe 2

Wie in Tabelle 6.16 zu sehen, profitieren alle 3 Queries recht deutlich von den Indizes. Die deutlichste Änderung gibt es jedoch bei Query 2.3, welcher im folgenden im Detail untersucht werden soll.

	q2.1	q2.2	q2.3
Row[ms]	139	111	102
Row Index[ms]	73	16	4
Verbesserung[ms]	66	95	98
Verbesserung[%]	47.4%	85.5%	96.0%

Tabelle 6.16: Laufzeiten der Queries aus Gruppe 2.

Der Query Execution Plan für Query 2.3 ohne Index wird in Abbildung A11, der Plan für Query 2.3 mit Index in Abbildung A12, dargestellt.

Die Ursache für die große Verbesserung bei Query 2.3 lässt sich sehr leicht erkennen:

Anstatt die Tabellen Lineorder und Part erst über einen Table Scan zu durchsuchen und anschließend über einen Hash-Join zu verknüpfen,kann nun der einerseits der Index auf PartKey in Lineorder und Part für den Join genutzt werden. Außerdem kann der Index auf P_Brand genutzt werden, um Part schneller zu filtern.

Die Laufzeit für diesen einzelnen Schritt kann von 169,7ms auf 2ms reduziert werden.

6.4.4 Analyse der Queries aus Gruppe 4

Wie in Tabelle 6.18 zu sehen, wird Query 4.3 sogar schneller durch die Indizes. Die beiden anderen Queries werden jedoch deutlich langsamer. Die deutlichste Änderung gibt es bei Query 4.1, welcher im folgenden im Detail untersucht werden soll.

	q4.1	q4.2	q4.3
Row[ms]	200	175	125
Row Index[ms]	421	320	78
Verbesserung[ms]	-221	-145	47
Verbesserung[%]	-110%	-82,8%	37,6%

Tabelle 6.18: Laufzeiten der Queries aus Gruppe 4.

Der Query Execution Plan für Query 4.1 ohne Index wird in Abbildung A13, der Plan für Query 4.1 mit Index in Abbildung A14, dargestellt.

Auf den ersten Blick scheinen die Indizes Wirkung zu zeigen: Anstatt Table Scans in Kombination mit Hash-Joins zu Nutzen werden Index Search und Index Join verwendet.

Allerdings wird die Bedingung auf Custkey immer noch über einen Hash Join erfüllt. Bei der Version ohne Index sind die Treffermengen 400 und 1.2 Millionen. Mit Index sind die Treffermengen 6000 und 1.1 Millionen.

Für die durchschnittliche Laufzeit des Hash Joins gilt:[9]

$$\mathcal{O}(m+n) \tag{6.1}$$

Die Laufzeiten sollten also in etwa gleich sein, wenn der Join ohne Index nicht sogar langsamer sein sollte.

Trotzdem benötigt der Join ohne Index nur 168ms, mit Index werden 283ms benötigt. Warum die Laufzeiten dennoch so stark voneinandern abweichen, ist nicht klar. Möglicherweise ist die Hashtabelle auf die Treffermenge 6000 Einträgen zu groß, um komplett im Hauptspeicher zu liegen. Überprüft wurde dies jedoch nicht.

Der Unterschied ist beim Hash-Join zwar am auffälligsten, allerdings sind auch viele andere Schritte, wie zum Beispiel die Suche auf Lineorder, bei der Version ohne Index schneller, was sich in Summe in dem großen Laufzeitunterschied niederschlägt.

6.4.5 Fazit für Rowstore

Auch Rowstores können durch geschickt gewählte Indizes deutlich beschleunigt werden. Sie profitieren zwar nicht ganz so stark wie Columstores, werden aber doch merklich schneller. Allerdings scheint es auch Fälle zu geben, in denen Indizes deutliche Verschlimmbesserungen darstellen. Von blind angelegten Indizes ist also abzuraten.

6.5 Fazit

Sowohl Columnstores als auch Rowstores könen durch Indizes nochmals teils deutlich beschleunigt werden. Bei Rowstores gab es jedoch auch Fälle, wo die Indizes den Query deutlich verlangsamten. Doch auch mit Indizes liegen Rowstores immer noch weit hinter Columnstores.

6.6 Auswirkung der OLAP-Engine

Bei der Analyse von Query 1.2 ist aufgefallen, dass der Query manchmal über die OLAP-Engine ausgeführt[1] wird. Die OLAP-Engine ist speziell für „Analytical Views“, die als Columnstore im Star-Schema vorliegen, gedacht.[7, 14] Beim Star-Schema Benchmark liegen die Daten definitv im Star-Schema vor.

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. Warum die OLAP-Engine scheinbar zufällig verwendet wurde, ist unklar.

Soll gezielt die OLAP-Engine genutzt werden, so kann dies durch den Hint „USE_OLAP_PLAN“ erzwungen werden.[6, 11] Um die Auswirkung der OLAP-Engine genauer zu untersuchen, wurden die Benchmarks mit/ohne Indidzes, siehe Abschnitt 6.4, jeweils mit und ohne den OLAP-Hint ausgeführt.

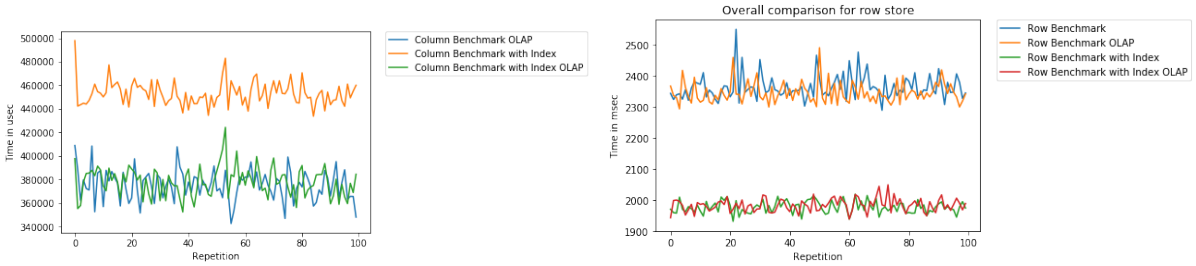


Abbildung 6.9: Vergleich der Gesamtlaufzeit mit OLAP-Hint. n=100

6.6.1 Untersuchung für Columnstore

Beim Vergleich der verschiedenen Kombinationen für Columstores war das Ergebnis eindeutig: Die OLAP-Engine schlägt die normale Column-Engine um Längen.

Im Schnitt ist die OLAP-Engine 38 ms schneller, als die Column-Engine mit Indizes. Lässt man die Indizes weg wird der Unterschied noch größer, da die Column-Engine langsamer wird. Die OLAP-Engine scheint von den Inidzes nicht beeinflusst zu werden, siehe Tabelle 6.19.

[1] Um zu sehen, welche Engine verwendet wird, muss anstatt „Visualize Plan“ die Option „Explain Plan“ gewählt werden.

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.

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

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

Tabelle 6.19: Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Columnstore.

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 6.21: Laufzeit jeder Benchmarkgruppe für Columnstore mit Index, testweise mit OLAP-Hint. n=100

6.6.2 Untersuchungen für Rowstore

Bei der Untersuchung der Ergebnisse für Rowstore und wie bereits in Abbildung 6.9 zu sehen, hat die OLAP-Engine keine Auswirkung auf die Laufzeit des Benchmarks.

Vermutlich liegt dies daran, dass die OLAP-Engine nur auf Analytical-Views, die im Columnstore gespeichert sind, ausgelegt ist. Überprüft wurde dies aber nicht.

Engine	No Index [ms]	Index [ms]
OLAP	2362	1976
Row	2344	1977

Tabelle 6.22: Durchschnitt der Gesamtlaufzeit mit und ohne OLAP-Engine bei Rowstore.

6.6.3 Fazit

Durch Nutzung der OLAP-Engine kann man eventuelle Beschleunigungen durch Indizes, zumindest für Columnstores, nochmals deutlich überbieten. 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.[12] Alternativ könnte man seine Anfragen „formgerecht“ als Analytical-View aufbauen, um die OLAP-Engine nutzen zu können.[2, 14]

Für Rowstores kann die OLAP-Engine leider nicht verwendet werden.

6.7 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. In der Analyse werden wird zuerst auf den Columnstore eingegangen und anschließend vergleichend auf den Rowstore.

6.7.1 Columnstore

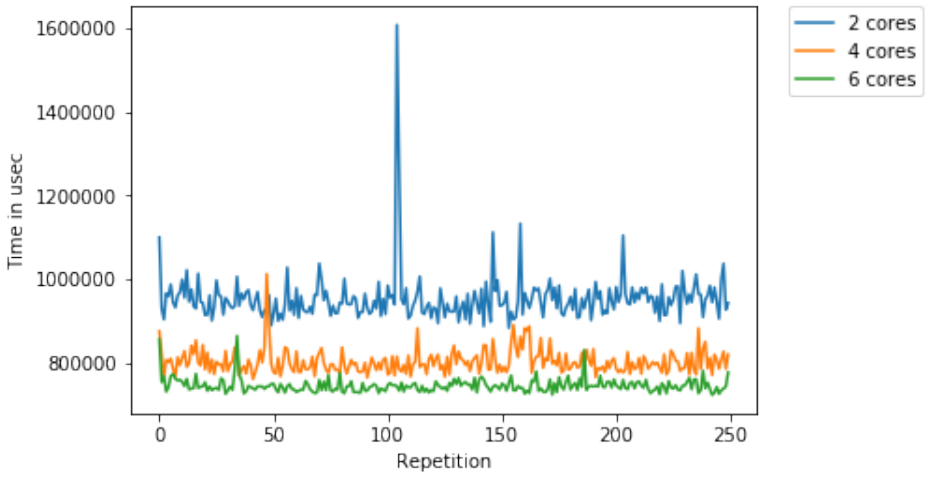


Abbildung 6.10: Benchmark im Columnstore unter Variation der CPU Kerne

Abbildung 6.10 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 6-Kerner gebildet und die langsameren Werte durch den zwei-Kerner. Werden der 6-Kerner(0.75 Sekunden) und der 2-Kerner(0.95 Sekunden) anhand ihren arithmetischen Mitteln miteinander verglichen so ergibt sich eine prozentuale Differenz von 26.7%. Der 4-Kerner hingegen erreicht eine durchschnittliche Ausführungszeit von 0,8 Sekunden was ihm eine Differenz von 7% im Vergleich zum 6-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 eine quadratische Abnahme der Ausführungszeit abhängig zur Anzahl der CPU-Kerne schliessen.

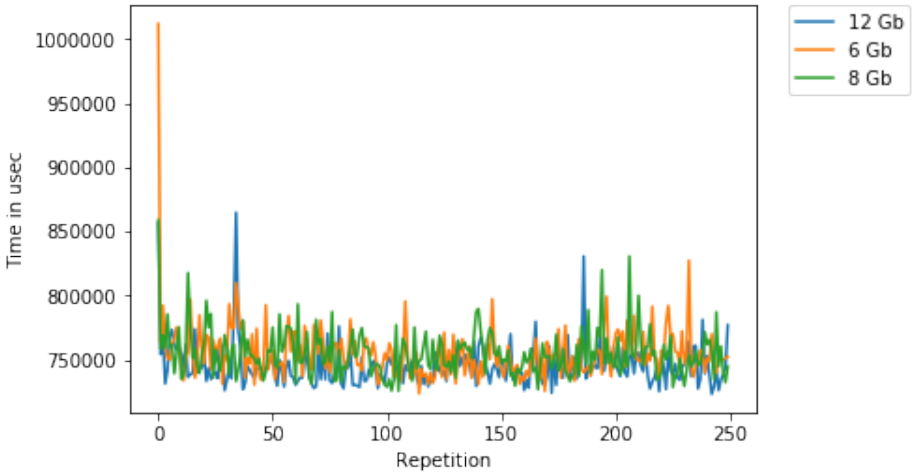


Abbildung 6.11: Benchmark im Columnstore unter Variation des RAMs

Abbildung 6.11 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 vorallem darin begründet, dass bei reduziertem RAM externe Faktoren (Cache-Miss, Zugriffszeit) deutlicher zum tragen kommen.

6.7.2 Rowstore

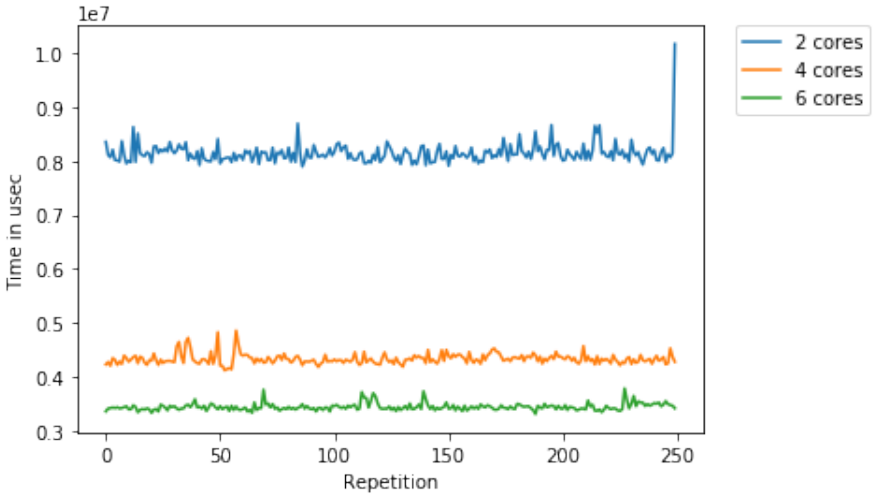


Abbildung 6.12: Benchmark im Rowstore unter Variation der CPU Kerne

Abbildung 6.12 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 2-Kerner liegt mit einer durschnittlichen Ausführungszeit von 8,2 Sekunden 137% über der Ausführungszeit des acht-Kerners, welcher nur 3,4 Sekunden im Schnitt braucht. Die Abweichung des 6-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 irrelevanter 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 ensteht eine erhöhte Relevanz der CPU für die Ausführungszeit als beim Columnstore.

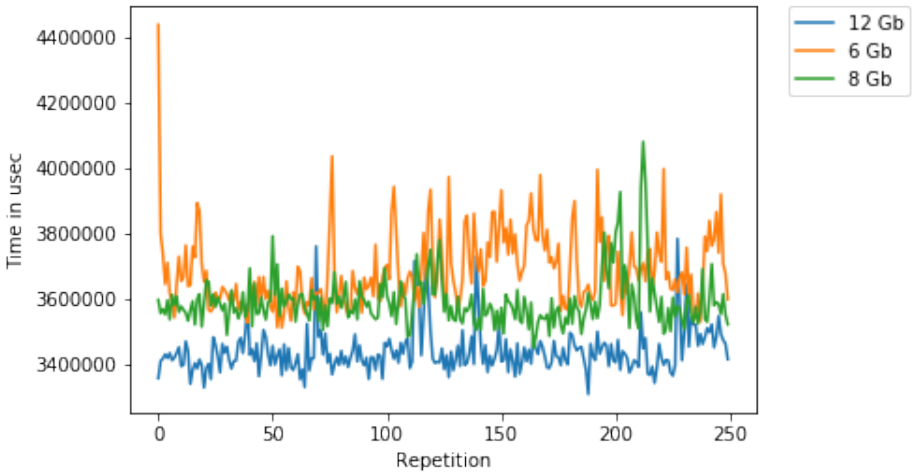


Abbildung 6.13: Benchmark im Rowstore unter Variation des RAMs

Abbildung 6.13 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 Columstore 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 nur bei ausreichend RAM durchgeführt werden.

7 Fazit

Generell lässt sich sagen, dass der Columnstore für die im SSBM Benchmark geschaffenen Bedingungen deutlich besser geeignet ist. In Zahlen bedeutet dies, dass der Columnstore im Schnitt mehr als 4 mal so schnell ist, wie der Rowstore.

Die exemplarische Analyse der Query Execution Pläne für Query 4.3 hat gezeigt, dass der große zeitliche Unterschied zwischen Row- und Columnstore darauf zurückzuführen ist, dass die Queries im Columnstore paralellisiert ablaufen. Hier gibt es außerdem eine klare Trennung zwischen Filter und Join, beim Rowstore ist dies enger miteinander verknüpft.

Durch Indizes kann der Benchmark in beiden Fällen nochmals teils deutlich beschleunigt werden, allerdings gibt es besonders beim Rowstore auch Fälle, wo bestimmte Queries durch die Indizes verlangsamt wurden. Indizes sollten also bewusst eingesetzt und auf ungewünschte Nebeneffekte geprüft werden.

Außerdem hat sich gezeigt, dass die optimierte OLAP-Engine deutlich schneller ist, also die schnellste Variante der Column-Engine. In der Praxis kann es von Vorteil sein, Abfragen als Analytical Views zu realisieren, um diese Engine nutzen zu können. Für Rowstores ist die Engine leider nicht verfügbar.

Des Weiteren hat sich gezeigt, dass der Columnstore von mehr CPU-Kernen profitiert, da hier sehr stark parallelisiert werden kann. Die Messwerte lassen auf einen quadratischen Abnahme der Ausführungszeit abhängig zur Anzahl der CPU-Kerne schliessen. Durch reduzieren des Hauptspeicher nahm die Stabilität des Benchmarks stark ab, was vermutlich darauf zurückzuführen ist, dass bei reduziertem RAM externe Faktoren (Cache-Miss, Zugriffszeit) zum tragen kommen.

Beim Rowstore hat sich gezeigt, dass hier sowohl CPU als auch RAM von größerer Bedeutung für die Laufzeit sind. Während im Columnstore automatisch Optimierungen, wie Indexierung und Kompression stattfinden, können diese im Rowstore nur bei ausreichend RAM durchgeführt werden. Außerdem wird im Rowstore die Performance durch wenige Kerne aufgrund der fehlenden Parallelität umso mehr eingeschränkt, da Lesezugriffe auf die Ausführung der CPU warten müssen und umgekehrt.

Zusammenfassend lässt sich sagen, dass die oft im Netz zu findende Empfehlung, hauptsächlich auf Columnstores zu setzen, durch unsere Ergebnisse unterstützt wird.

8 Autoren - Wer hat was geschrieben

Name	Matrikelnummer	Hat geschrieben
Jendrik Jordan	7344208	Kapitel 2, Kapitel 3, Abschnitt 5.1, Abschnitt 5.2, Abschnitt 5.3, Abschnitt 6.7
Arwed Mett	4278042	Abschnitt 5.4, Abschnitt 6.1, Abschnitt 6.2
Simon Oswald	6594512	Abstract, Abschnitt 6.4, Abschnitt 6.6, Kapitel 7
Dominic Steinhauser	1807718	Abstract, Kapitel 1, Kapitel 4, Abschnitt 6.3

Literatur

[1] *Column Vs Row Data Storage*. URL: http://saphanatutorial.com/column-data-storage-and-row-data-storage-sap-hana/.

[2] *Exploit Underlying Engine*. Mai 2017. URL: https://goo.gl/lT2U96.

[3] *GNU Bash*. Mai 2017. URL: https://www.gnu.org/software/bash/.

[4] Mohan Govindarajan. *Understanding Row Store and Column Store?* 2016. URL: https://archive.sap.com/discussions/thread/3174364.

[5] Ankur Goyal. *Should You Use a Rowstore or a Columnstore?* 2016. URL: http://blog.memsql.com/should-you-use-a-rowstore-or-a-columnstore/.

[6] *HINT Details*. Mai 2017. URL: http://help-legacy.sap.com/saphelp_hanaplatform/helpdata/en/4b/a9edce1f2347a0b9fcda99879c17a1/content.htm.

[7] *Introduction to SAP HANA Modeling*. Mai 2017. URL: http://saphanatutorial.com/sap-hana-modeling/.

[8] *SQL:2011*. Standard. Geneva, CH: International Organization for Standardization, 2011.

[9] *Joinalgorithmen*. Mai 2017. URL: https://de.wikipedia.org/wiki/Joinalgorithmen.

[10] *The JavaScript Object Notation (JSON) Data Interchange Format*. Standard. Internet Engineering Task Force, 2014.

[11] *SAP HANA Important Hints*. Mai 2017. URL: https://www.stechies.com/important-hints-related-sap-hana/.

[12] *#SAPHANA Query Hints - Looking for documentation*. Mai 2017. URL: https://archive.sap.com/discussions/thread/3277920.

[13] *Spaltenorientierte Datenbank*. Mai 2017. URL: https://de.wikipedia.org/wiki/Spaltenorientierte_Datenbank.

[14] *The SAP HANA Engines?* Mai 2017. URL: https://archive.sap.com/discussions/thread/3340726.

[15] *TPC BENCHMARKTM H*. Techn. Ber. Version 2.17.2. 2017. URL: http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf.

[16] *Transaction Processing Performance Council*. 2015. URL: https://de.wikipedia.org/wiki/Transaction_Processing_Performance_Council.

Anhang

```
DROP TABLE  customer;

CREATE COLUMN TABLE  customer (
  C_CUSTOMERKEY INTEGER,
  C_Name varchar(25),
  C_Address varchar(25),
  C_City varchar(10),
  C_Nation varchar(15),
  C_Region varchar(12),
  C_Phone varchar(15),
  C_MktSegment varchar(10),
  PRIMARY KEY ("C_CUSTOMERKEY")
);

DROP TABLE  part;

CREATE COLUMN TABLE  part
(
  P_PartKey integer,
  P_Name varchar(25),
  P_MFGR varchar(10),
  P_Category varchar(10),
  P_Brand varchar(15),
  P_Colour varchar(15),
  P_Type varchar(25),
  P_Size tinyint,
  P_Container char(10),
  PRIMARY KEY (P_PartKey)
);



DROP TABLE  supplier;

CREATE COLUMN TABLE  supplier (
  S_SuppKey integer,
  S_Name char(25),
  S_Address varchar(25),
  S_City char(10),
  S_Nation char(15),
  S_Region char(12),
```

```
  S_Phone char(15),
  PRIMARY KEY(S_SuppKey)
);



DROP TABLE  dim_date;

CREATE COLUMN TABLE  dim_date
(
  D_DateKey integer,
  D_Date char(18),
  D_DayOfWeek char(9),
  D_Month char(9),
  D_Year smallint,
  D_YearMonthNum integer,
  D_YearMonth char(7),
  D_DayNumInWeek tinyint,
  D_DayNumInMonth tinyint,
  D_DayNumInYear smallint,
  D_MonthNumInYear tinyint,
  D_WeekNumInYear tinyint,
  D_SellingSeason char(12),
  D_LastDayInWeekFl tinyint,
  D_LastDayInMonthFl tinyint,
  D_HolidayFl tinyint,
  D_WeekDayFl tinyint,
  PRIMARY KEY(D_DateKey)
);

DROP TABLE lineorder;

CREATE COLUMN TABLE  lineorder
(
  LO_OrderKey bigint not null,
  LO_LineNumber tinyint not null,
  LO_CustKey int not null,
  LO_PartKey int not null,
  LO_SuppKey int not null,
  LO_OrderDateKey int not null,
  LO_OrderPriority varchar(15),
  LO_ShipPriority char(1),
  LO_Quantity tinyint,
  LO_ExtendedPrice decimal,
  LO_OrdTotalPrice decimal,
  LO_Discount decimal,
  LO_Revenue decimal,
```

```
  LO_SupplyCost decimal,
  LO_Tax tinyint,
  LO_CommitDateKey integer not null,
  LO_ShipMode varchar(10)
);
```

Listing 1: Schema Columnstore

```
DROP TABLE  customer;

CREATE ROW TABLE  customer (
  C_CUSTOMERKEY INTEGER,
  C_Name varchar(25),
  C_Address varchar(25),
  C_City varchar(10),
  C_Nation varchar(15),
  C_Region varchar(12),
  C_Phone varchar(15),
  C_MktSegment varchar(10),
  PRIMARY KEY ("C_CUSTOMERKEY")
);

DROP TABLE  part;

CREATE ROW TABLE  part
(
  P_PartKey integer,
  P_Name varchar(25),
  P_MFGR varchar(10),
  P_Category varchar(10),
  P_Brand varchar(15),
  P_Colour varchar(15),
  P_Type varchar(25),
  P_Size tinyint,
  P_Container char(10),
  PRIMARY KEY (P_PartKey)
);



DROP TABLE  supplier;

CREATE ROW TABLE  supplier (
  S_SuppKey integer,
  S_Name char(25),
  S_Address varchar(25),
  S_City char(10),
  S_Nation char(15),
```

```
  S_Region char(12),
  S_Phone char(15),
  PRIMARY KEY(S_SuppKey)
);



DROP TABLE  dim_date;

CREATE ROW TABLE  dim_date
(
  D_DateKey integer,
  D_Date char(18),
  D_DayOfWeek char(9),
  D_Month char(9),
  D_Year smallint,
  D_YearMonthNum integer,
  D_YearMonth char(7),
  D_DayNumInWeek tinyint,
  D_DayNumInMonth tinyint,
  D_DayNumInYear smallint,
  D_MonthNumInYear tinyint,
  D_WeekNumInYear tinyint,
  D_SellingSeason char(12),
  D_LastDayInWeekFl tinyint,
  D_LastDayInMonthFl tinyint,
  D_HolidayFl tinyint,
  D_WeekDayFl tinyint,
  PRIMARY KEY(D_DateKey)
);

DROP TABLE lineorder;

CREATE ROW TABLE  lineorder
(
  LO_OrderKey bigint not null,
  LO_LineNumber tinyint not null,
  LO_CustKey int not null,
  LO_PartKey int not null,
  LO_SuppKey int not null,
  LO_OrderDateKey int not null,
  LO_OrderPriority varchar(15),
  LO_ShipPriority char(1),
  LO_Quantity tinyint,
  LO_ExtendedPrice decimal,
  LO_OrdTotalPrice decimal,
  LO_Discount decimal,
```

```
  LO_Revenue decimal,
  LO_SupplyCost decimal,
  LO_Tax tinyint,
  LO_CommitDateKey integer not null,
  LO_ShipMode varchar(10)
);
```

Listing 2: Schema Rowstore

```
CREATE INDEX idx_c_name ON customer(C_Name);
CREATE INDEX idx_c_city ON customer(C_City);
CREATE INDEX idx_c_region ON customer(C_Region);
CREATE INDEX idx_c_phone ON customer(C_Phone);
CREATE INDEX idx_c_mktsegment ON customer(C_MktSegment);

CREATE INDEX idx_p_name ON part(P_Name);
CREATE INDEX idx_p_mfgr ON part(P_MFGR);
CREATE INDEX idx_p_category ON part(P_Category);
CREATE INDEX idx_p_brand ON part(P_Brand);

CREATE INDEX idx_s_city ON supplier(S_City);
CREATE INDEX idx_s_name ON supplier(S_Name);
CREATE INDEX idx_s_phone ON supplier(S_Phone);

CREATE INDEX idx_lo_orderkey_lo_linenumber ON lineorder(LO_OrderKey,
    LO_LineNumber);
CREATE INDEX idx_lo_custkey ON lineorder(LO_CustKey);
CREATE INDEX idx_lo_suppkey ON lineorder(LO_SuppKey);
CREATE INDEX idx_lo_partkey ON lineorder(LO_PartKey);
CREATE INDEX idx_lo_orderdatekey ON lineorder(LO_OrderDateKey);
CREATE INDEX idx_lo_commitdatekey ON lineorder(LO_CommitDateKey);
```

Listing 3: Indizes hinzufügen

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

IMPORT FROM CSV FILE '/usr/sap/HXE/HDB90/work/customer.tbl' INTO "SYSTEM
    "."CUSTOMER"
WITH
record delimited by '\n'
field delimited by '|';

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

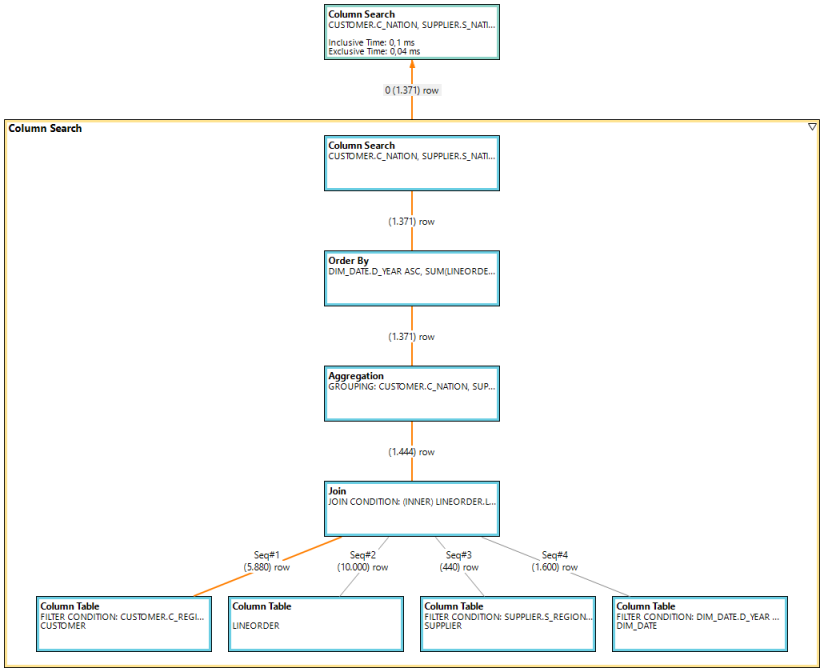


Abbildung A1: Q3.1 Execution Plan - Comlumn Store

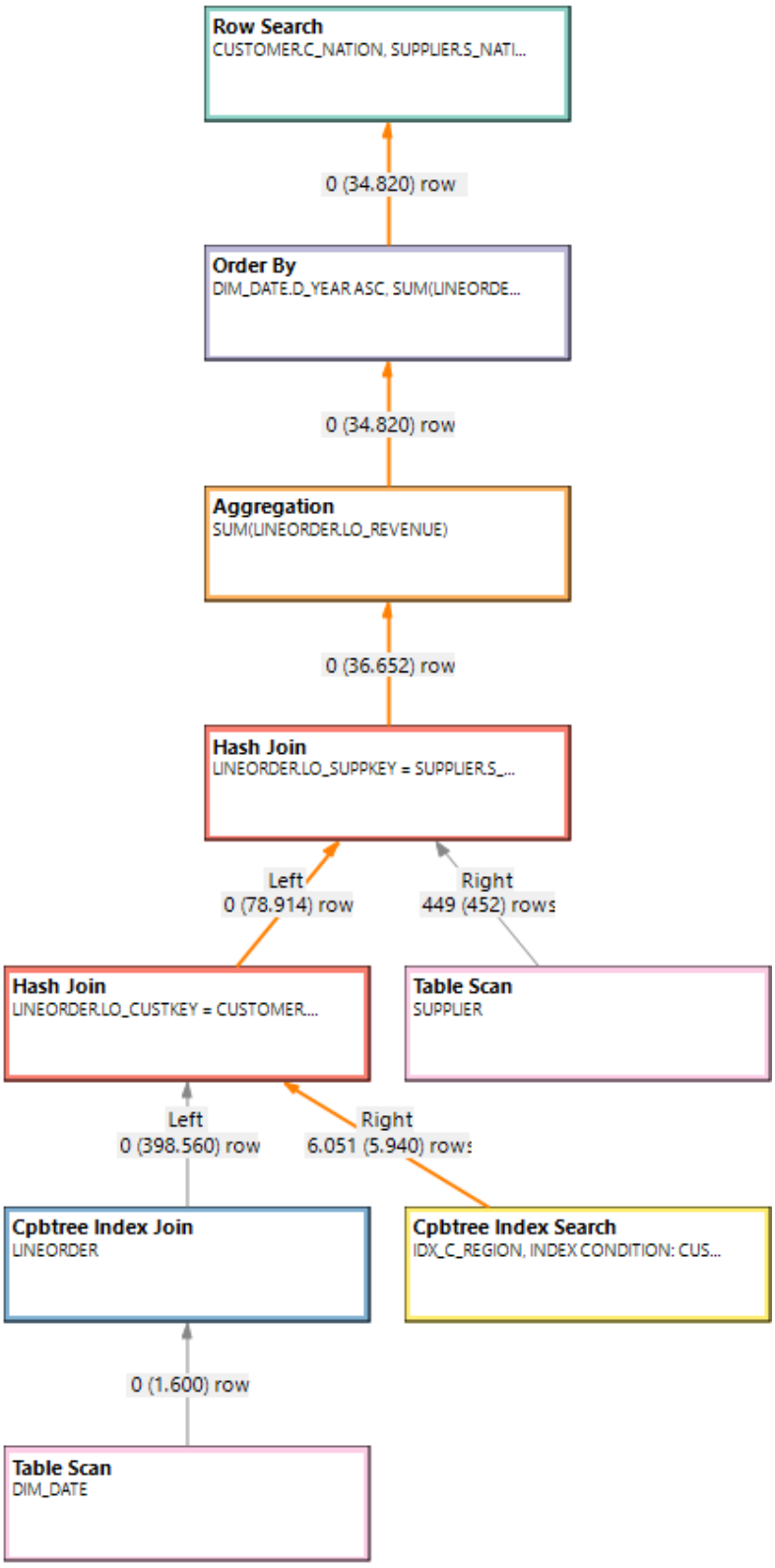


Abbildung A2: Q3.1 Execution Plan - Rowstore

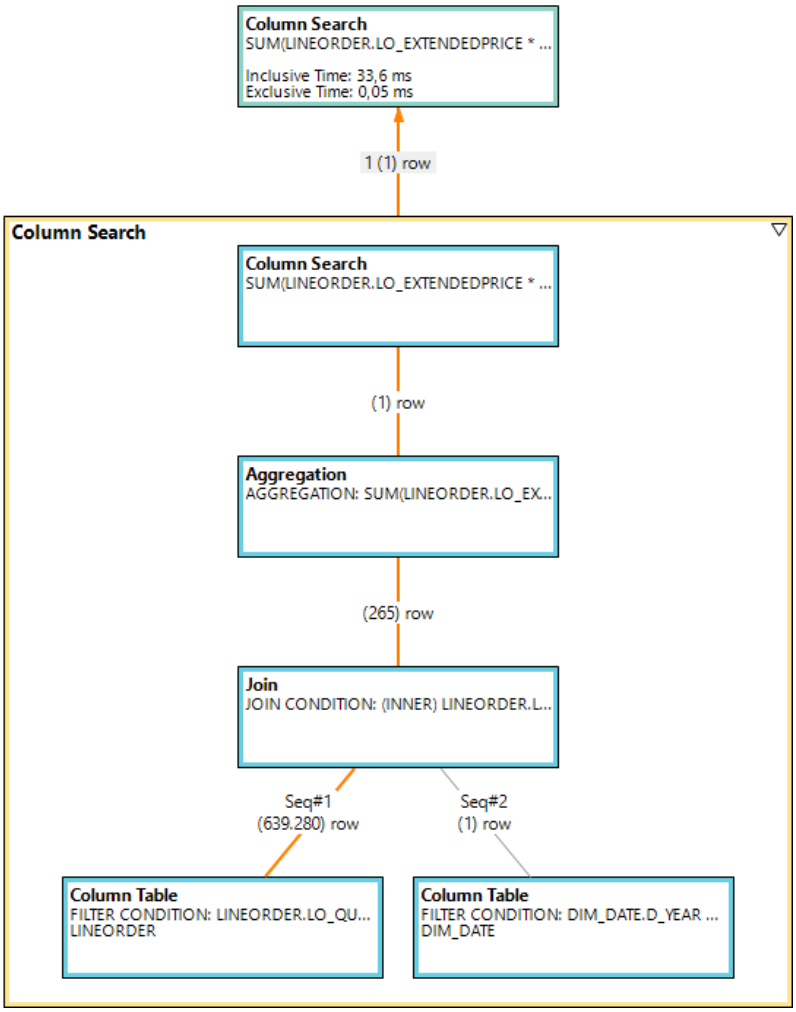


Abbildung A3: Q1.1 Execution Plan - Columnstore

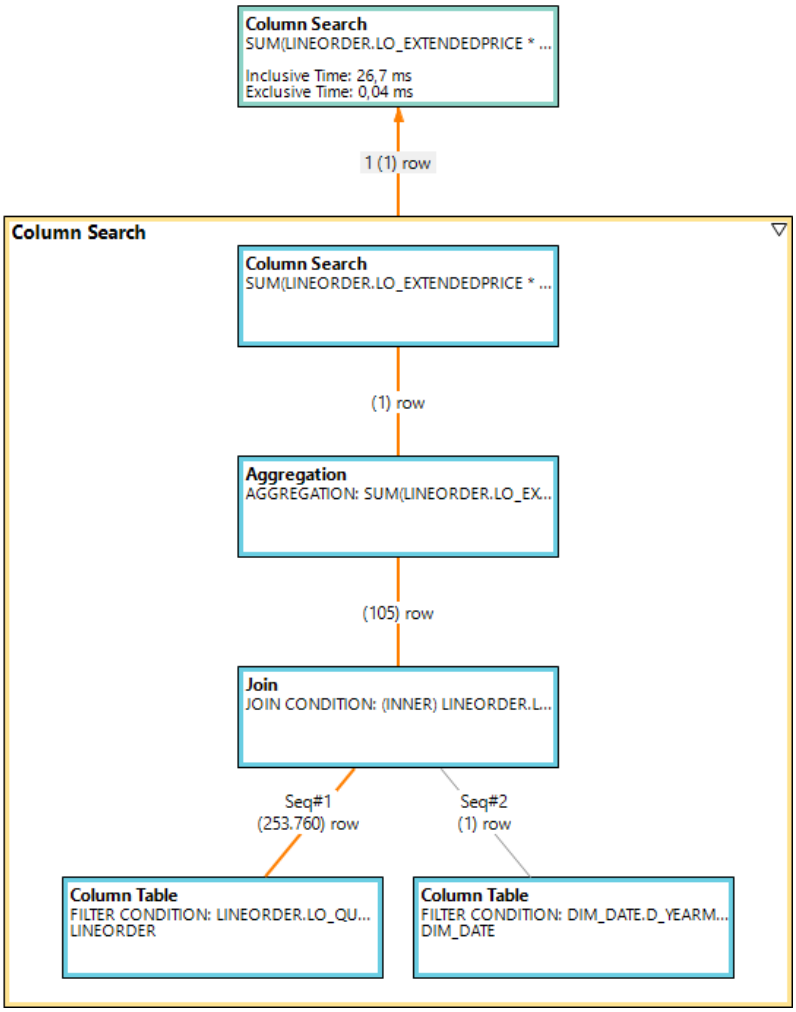


Abbildung A4: Q1.2 Execution Plan - Columnstore

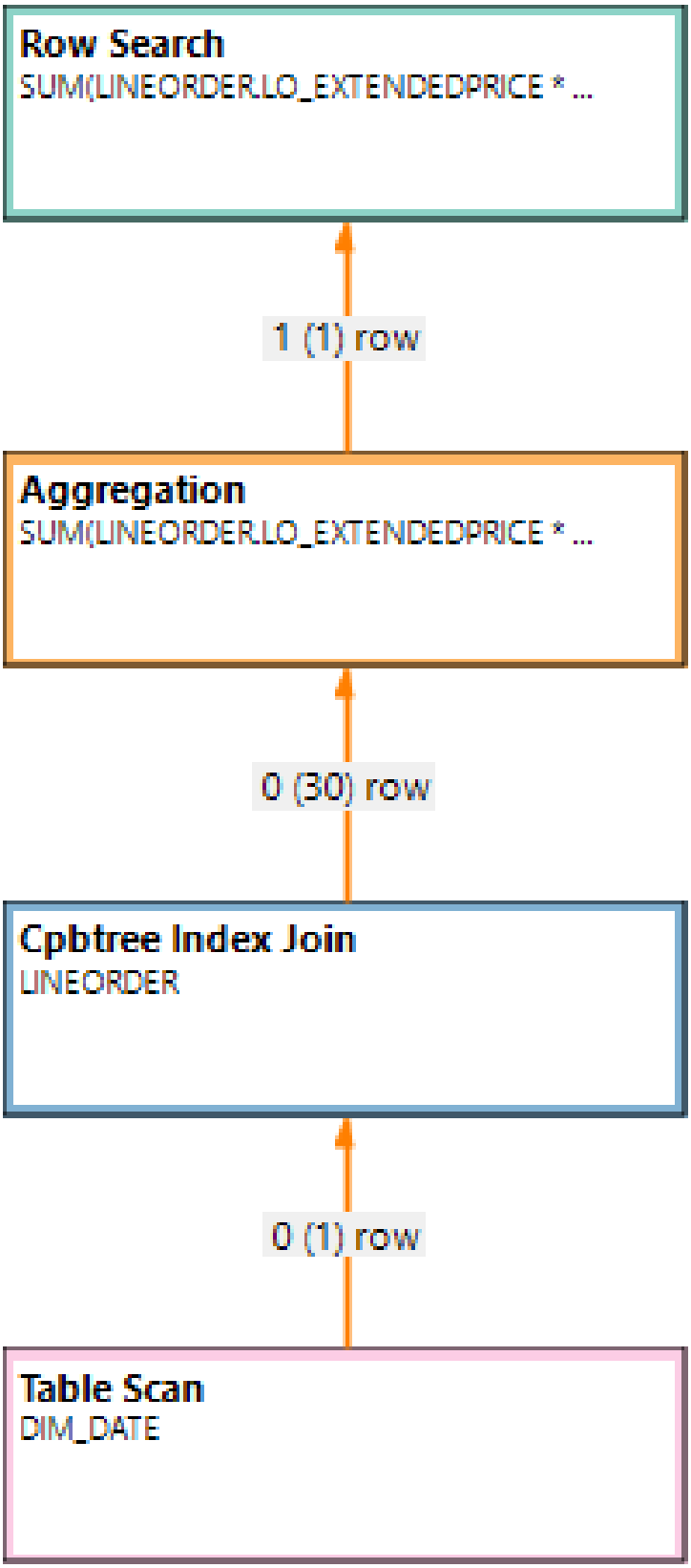


Abbildung A5: Q1.1 Execution Plan - Rowstore

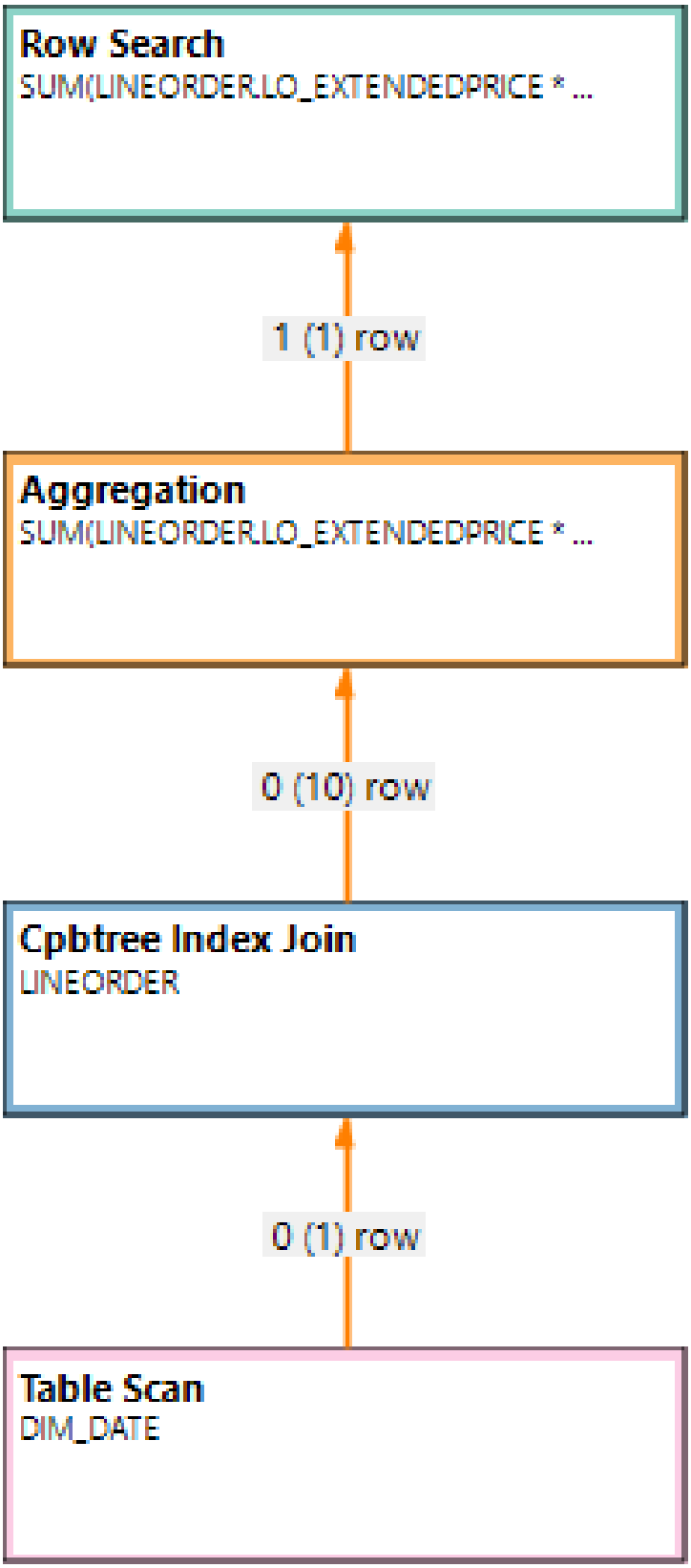


Abbildung A6: Q1.2 Execution Plan - Rowstore

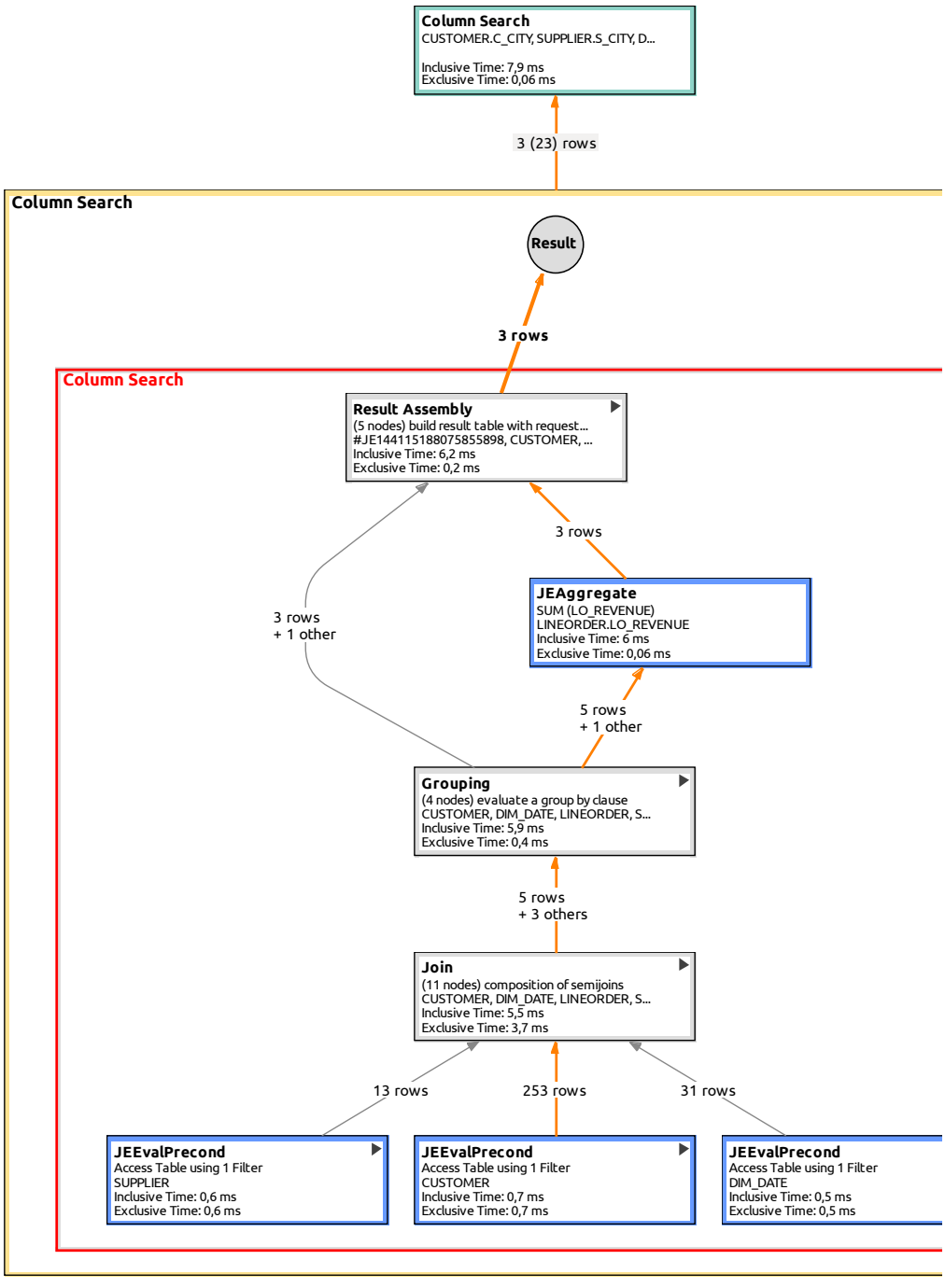


Abbildung A7: Q3.4 Execution Plan - Columnstore mit Indizes

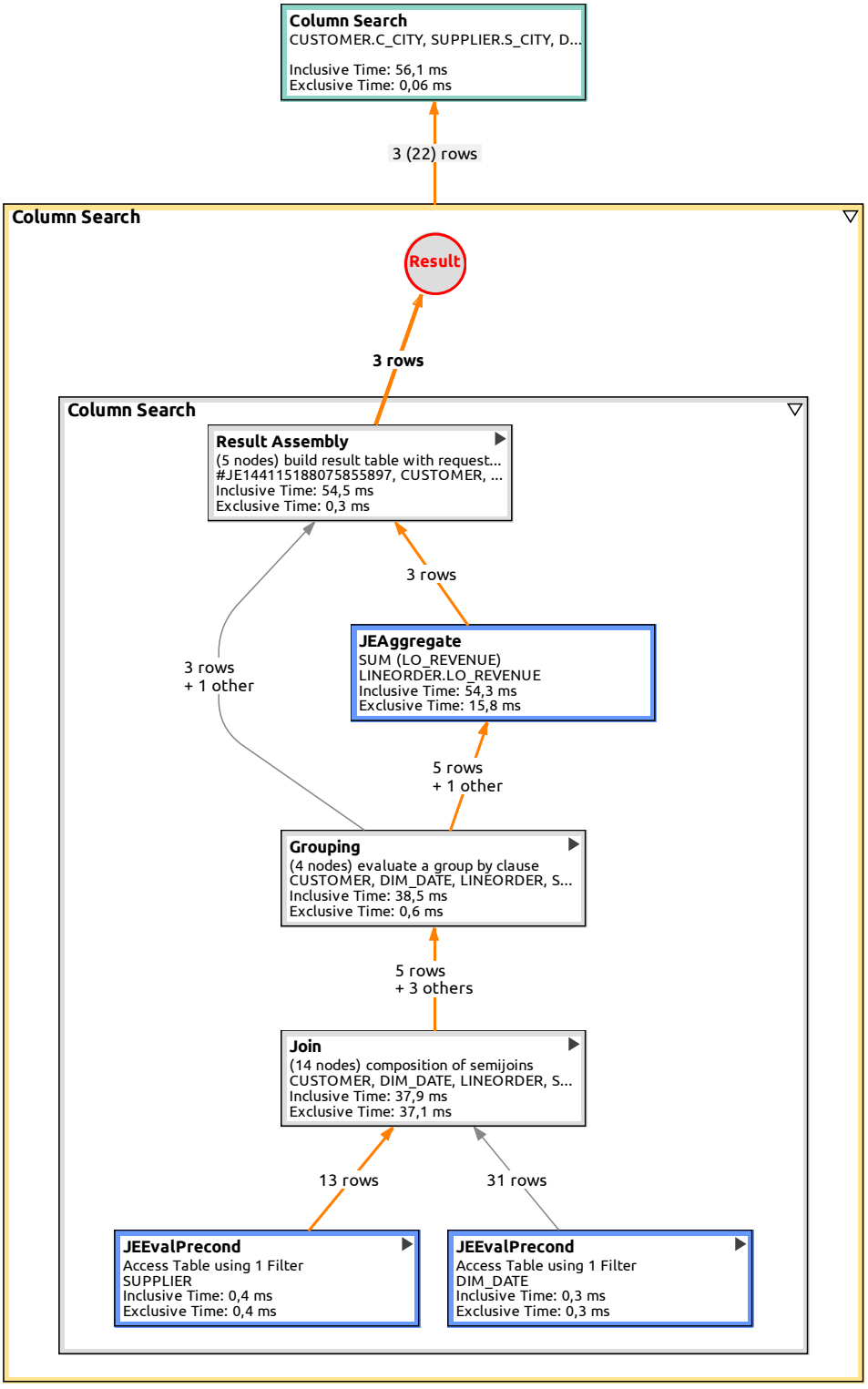


Abbildung A8: Q3.4 Execution Plan - Columnstore ohne Indizes

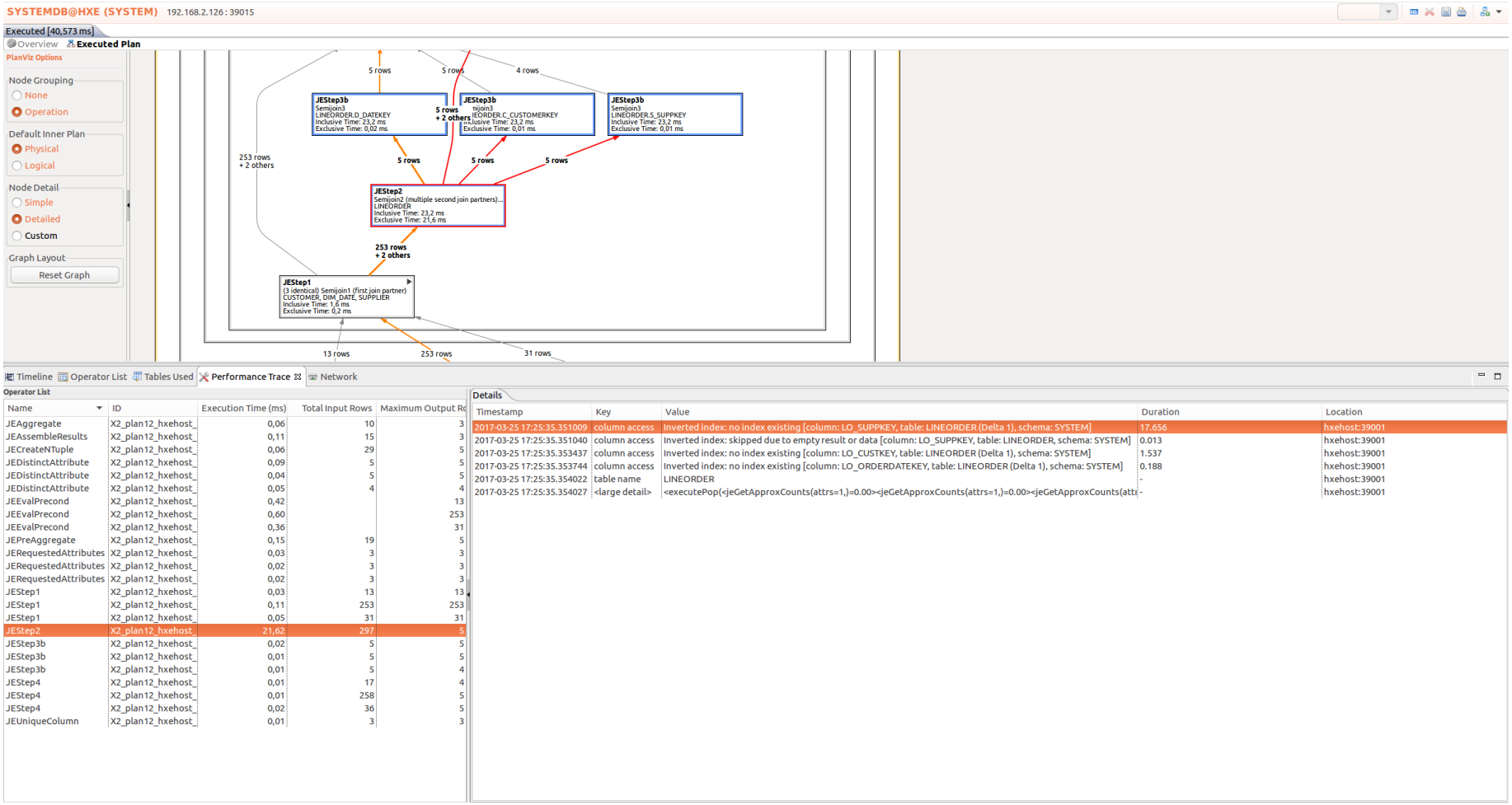


Abbildung A9: Execution Plan: Q3.4

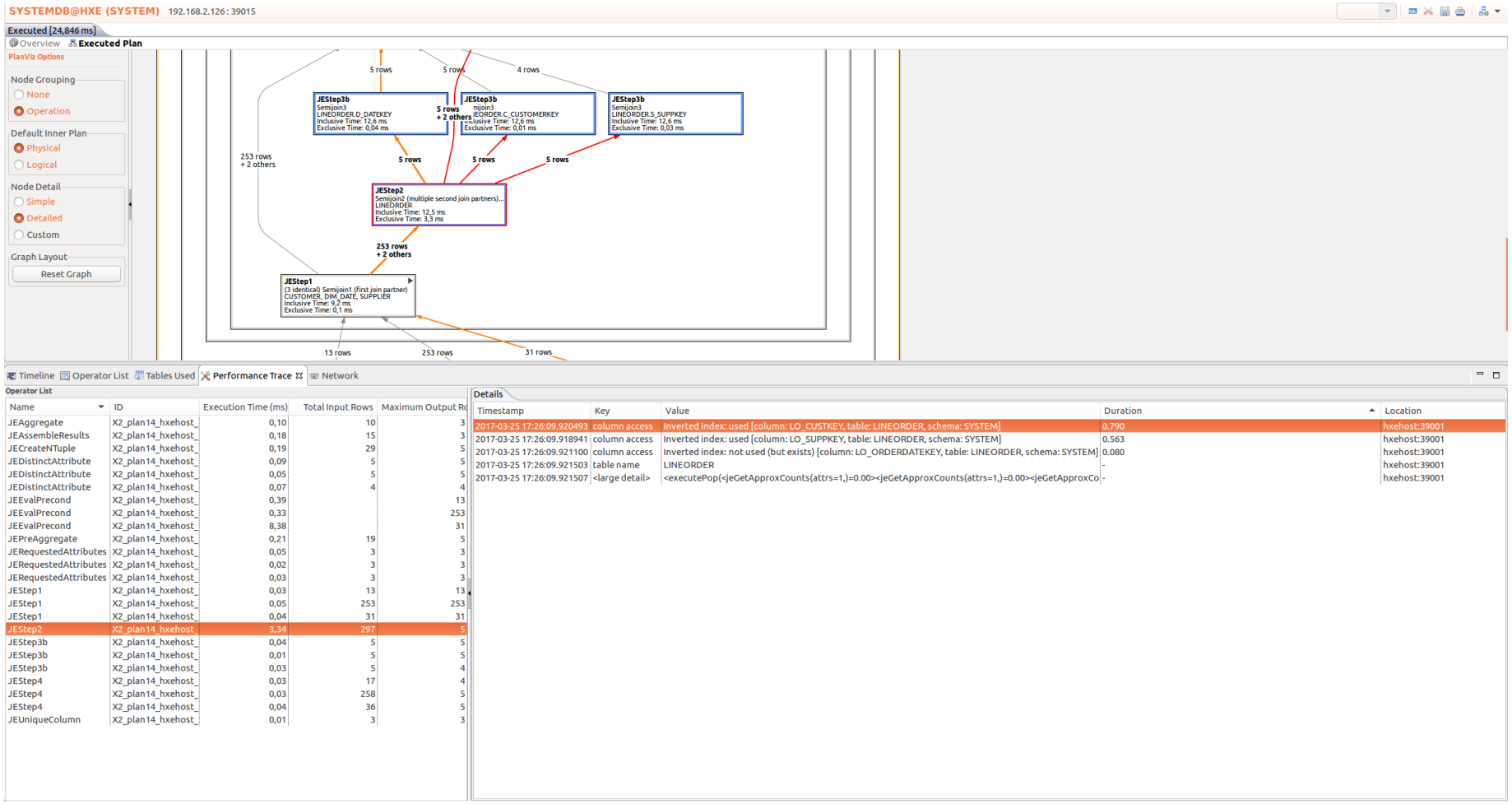


Abbildung A10: Execution Plan: Q3.4 with Index

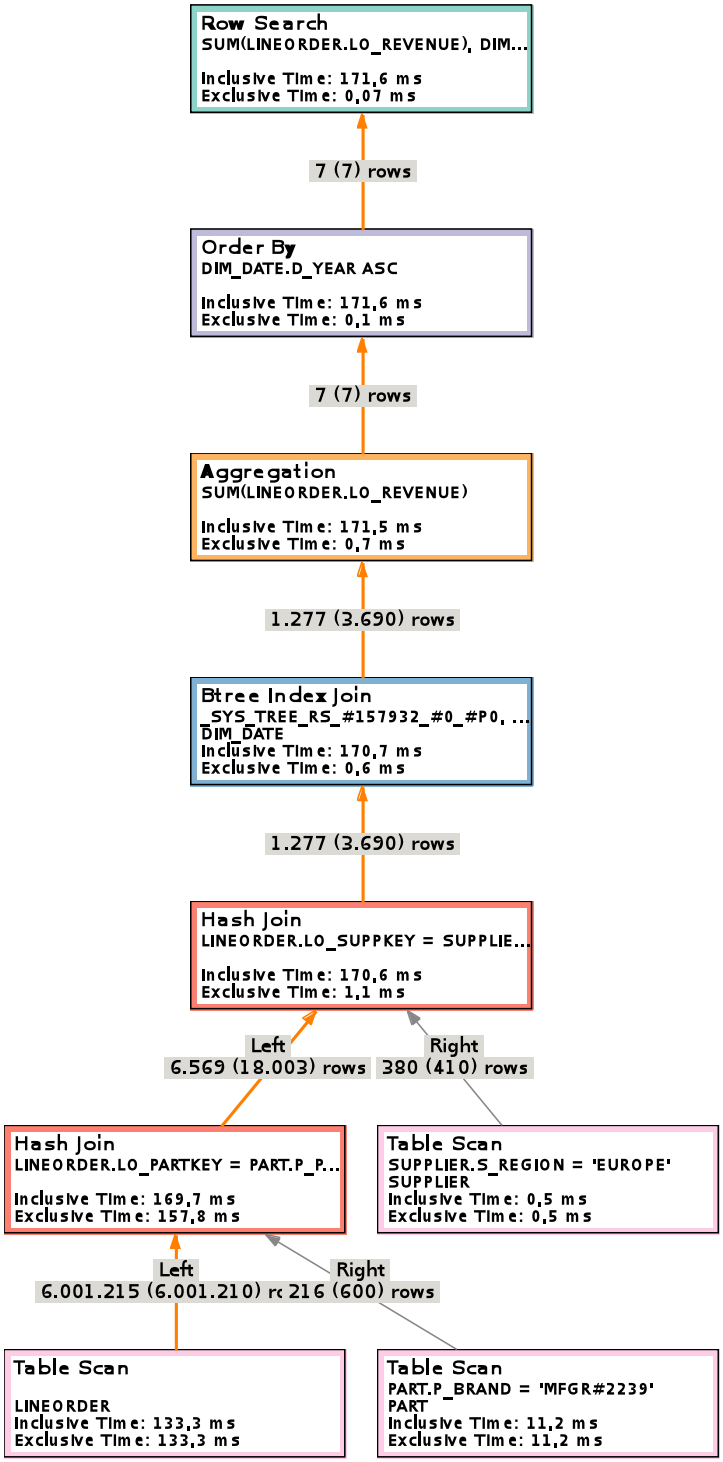


Abbildung A11: QEP für Query 2.3 ohne Index.

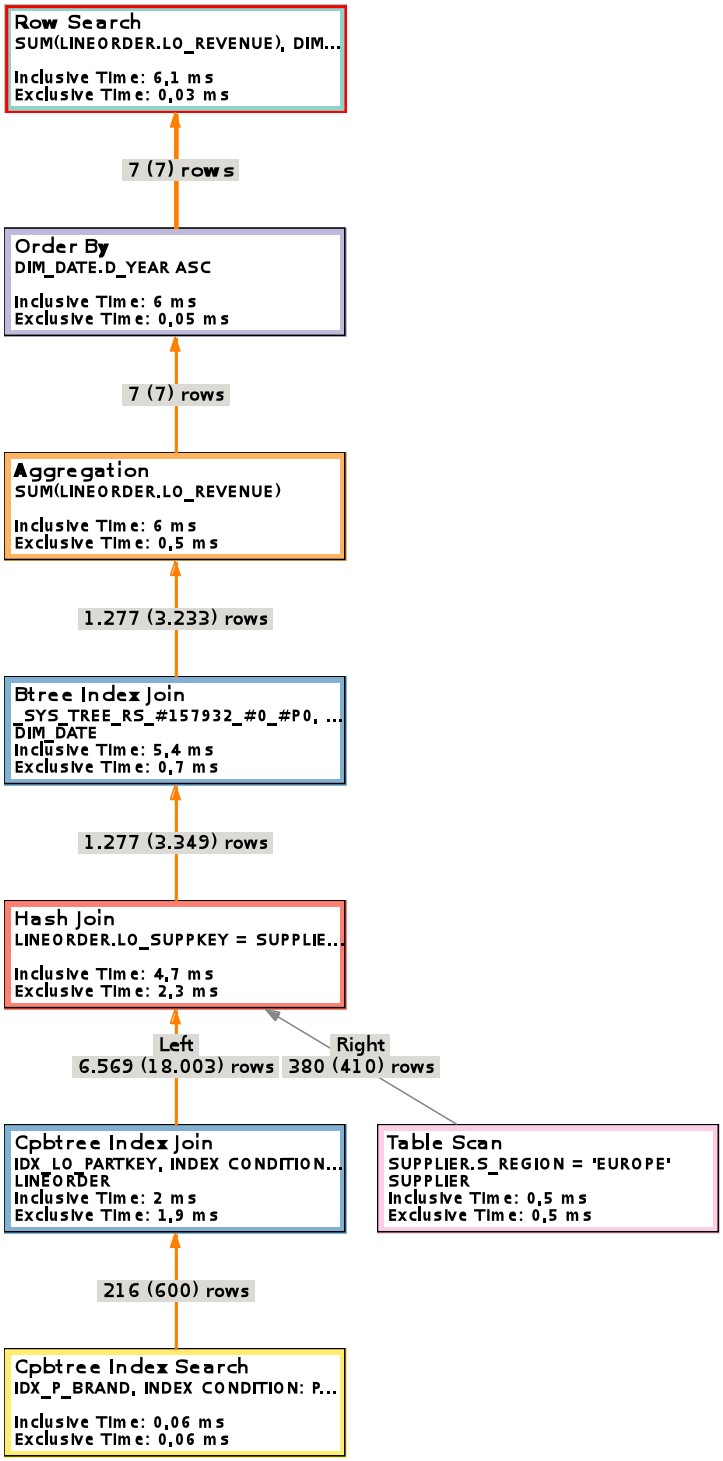


Abbildung A12: QEP für Query 2.3 mit Index.

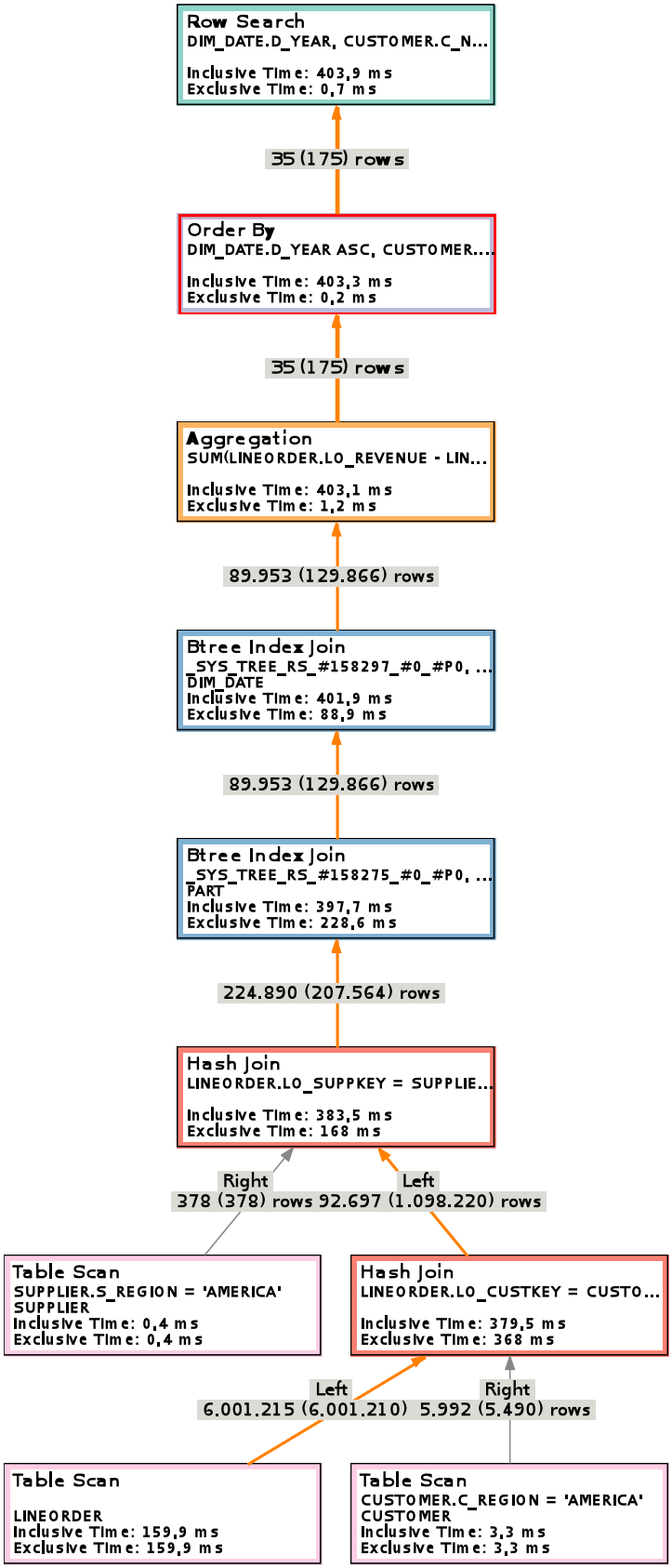


Abbildung A13: QEP für Query 4.1 ohne Index.

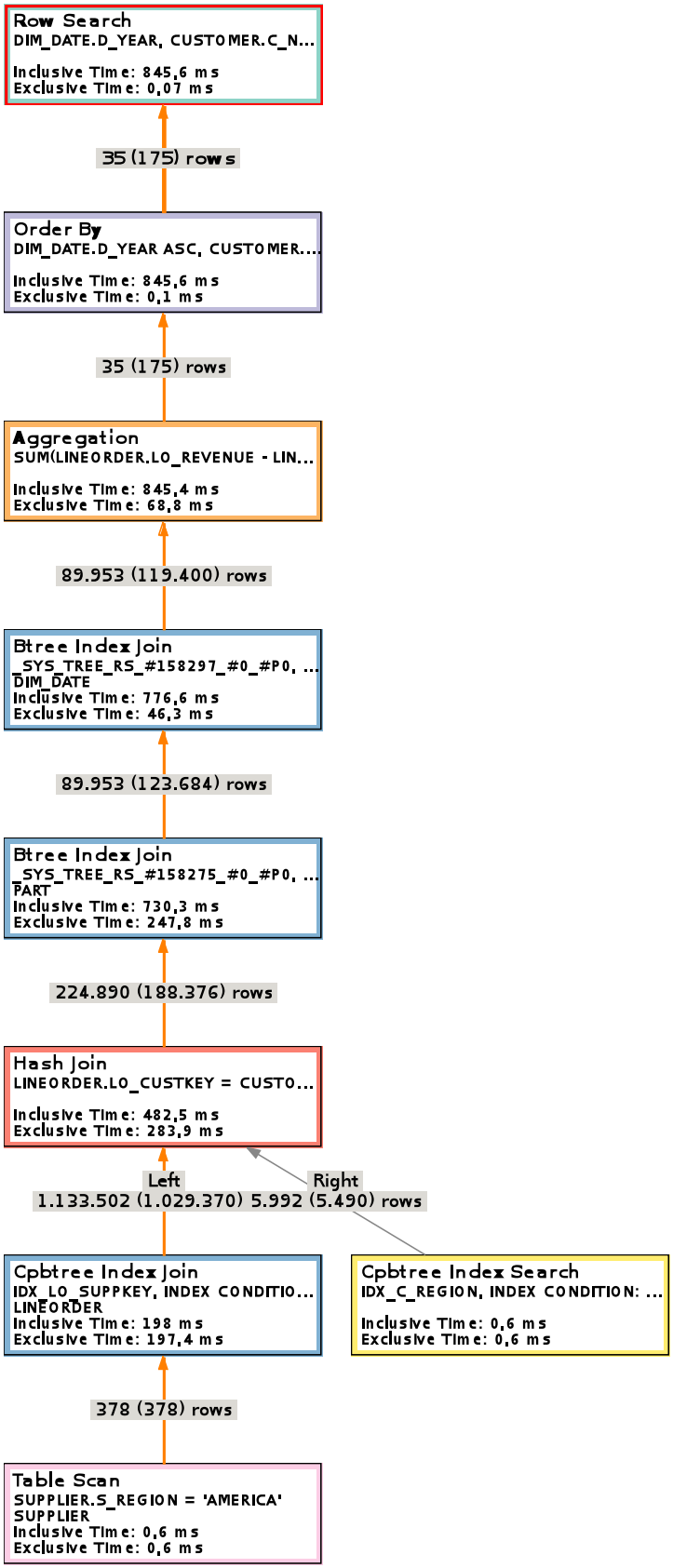


Abbildung A14: QEP für Query 4.1 mit Index.