

---

# **Star Schema Benchmark für SAP HANA**

Data Warehouse

JAN HOFMEIER, MARIUS JOCHHEIM, LION SCHERER,  
KRISTINA ALBRECHT

2018-03-06

## **Tabellenverzeichnis**

# Abbildungsverzeichnis

6.1	Benchmark-Cube . . . . .	12
-----	--------------------------	----

## List of Listings

# **1 SAP HANA**

## 2 Generell HANA als in-memory Datenbank

SAP Hana (Die High Performance Analytic Appliance) ist eine Entwicklungsplattform und besteht im Kern aus einer „in-memory“ Datenbank.

Transaktionen und Analysen werden auf einer einzigen, singulären Datenkopie im Hauptspeicher verarbeitet, anstatt die Festplatte als Datenspeicher zu benutzen. Dadurch ist es möglich sehr komplexe Abfragen und Datenbankoperationen mit sehr hohem Durchsatz auszuführen.

Hana verbindet OLTP, durch die SQL und ACID (Atomicity, Consistency, Isolation and Durability) Kompatibilität, und OLAP durch das „in-memory“ feature. Durch das ACID Prinzip ist die Datenbank geeignet um Unternehmensinterne Daten zu speichern. Es ist nicht nötig Datenanalysen über einen ETL Prozess an ein Datawarehouse weiterzuleiten. Komplexe Echtzeit Analysen [1] können nun direkt durch SAP Hana durchgeführt werden. Das erspart die erheblichen Kosten und vor allem Zeit.

Beim der „in-memory“ Technologie werden die Daten im Hauptspeicher anstatt auf elektromagnetischen Festplatten gespeichert. Antwortzeiten und Auswertungen können dadurch schneller als bei gewöhnlichen Festplatten durch den Prozessor vorgenommen werden. Dadurch, dass der Zugriff auf die Festplatte nun wegfällt, verkürzt sich die Datenzugriffszeit bis auf das Fünffache.

*img*

<https://intellipaat.com/blog/what-is-sap-hana/>

Um nun aber dem „D“ des ACID Prinzips gerecht zu werden reicht eine Speicherung im flüchtigen Hauptspeicher nicht. Für die Datensicherung müssen deshalb traditionelle Festplatten benutzt werden. Diese werden bei der reinen Analyse von Daten nicht berücksichtigt. Wenn Transaktionen getätigt werden, müssen die regelmäßig an das nicht flüchtige Speichermedium übergeben werden. Außerdem wird dort zu jeder Transaktion ein Protokolleintrag hinterlegt.

---

[1] <https://intellipaat.com/interview-question/sap-hana-interview-questions/>

2 <https://link.springer.com.ezproxy.dhbw-mannheim.de/book/10.1007%2F978-3-658-18603-6>

3 <https://www.sap.com/germany/products/hana.html#pdf-asset=2caaec36-847c-0010-82c7-eda71af511fa&page=3>

### **3 Row-Column based / Indizes**

Die Daten werden in der SAP Hana Datenbank in zwei verschiedenen Formaten abgelegt. Hierbei handelt es sich um die spalten- und zeilenorientierte Speicherung. Sollen beispielsweise transaktionale Prozesse (OLTP) durchgeführt werden, bietet sich die Verwendung der zeilenorientierten Speicherung an, da das Aktualisieren und Hinzufügen der Daten durch die Zeilen Anordnung vereinfacht wird.

Für Lesezugriffe ist diese Art der Speicherung nicht geeignet, da jede Zeile gelesen werden muss, was sehr unperformant ist. Es müssten Daten gelesen werden, die für die bestimmte Abfrage nicht von Relevanz sind. Daher werden Lesezugriffe und Analyseabfragen auf die spaltenorientierte Speicherung ausgeführt und somit wird nur auf die relevanten Daten zugegriffen. Dies hat eine enorme Performance zur Folge.

Durch die spaltenorientierte Speicherung erreicht man neben der Zugriffsbeschleunigung auch eine höhere Kompression der Daten. Die Daten können gut komprimiert werden, da Tabellenspalten häufig gleiche Werte enthalten.

Die Anzahl der Indizes kann erheblich reduziert werden. Bei der spaltenorientierten Speicherung kann jedes Attribut als Index verwendet werden. Da jedoch die gesamten Daten im Speicher vorhanden sind und die Daten einer Spalte alle aufeinanderfolgend gespeichert sind ist die Geschwindigkeit eines vollen sequentiellen Scans eines Attributs ausreichend in den meisten Fällen. Falls es nicht schnell genug ist können zusätzlich Indizes benutzt werden.

## 4 Komprimierungen und Referenzen

Warum Komprimierung?

Daten eignen sich. / CPU aufwand?

Bei der spaltenorientierten Speicherung ist es möglich Daten zu Komprimieren. Dadurch wird Speicherplatz gespart und Zugriffszeiten verringert. Es gibt zwei mögliche Komprimierungen:

### **Dictionary compression:**

Diese Methode wird auf alle Spalten angewandt. Alle verschiedenen Spaltenwerte werden aufeinanderfolgenden Zahlen zugeordnet. Anstatt nun die verschiedenen Werte zu speichern werden stattdessen die viel kleiner Zahlen gespeichert. Dadurch wird die Zahl der Datenzugriffe minimiert und es gibt weniger Cache Fehler, da mehrere Informationen in einer Cache-Line vorhanden sind. Außerdem ist es möglich Operationen direkt auf die komprimierten Daten auszuführen.

### **Advanced compression:**

Die einzelnen Zeilen selbst können durch verschiedene Komprimierungsmethoden weiter verkleinert werden. Dazu gehören:

#### **prefix encoding:**

Spalte enthält eine dominante Value / andere Values selten

- ein Wert wird sehr oft unkomprimiert gespeichert

datenset muss sortiert werden nach der Spalte mit der dominanten Value & der Attribut Vektor muss mit dem dominanten starten.

Zur Komprimierung sollte die dominante Value nicht jedes mal explizit gespeichert werden wenn sie auftritt.

Speichern der Nummer der Auftretungen der dominanten Value und eine Instanz der Value selbst im Attribut Vektor.

Prefix encoded Attribut Vektor enthält folgende Informationen:

Nummer der Auftretungen der dominanten Value



valueID der dominanten Value aus dem Dictionary  
valueIDs der fehlenden Values

### **run length encoding:**

Gut wenn ein Paar Werte mit hohem Aufkommen  
Sollte nach Werten sortiert sein für eine maximale Komprimierung  
Anstatt alle Werte einer Spalte zu Speichern werden lediglich 2 Vektoren gespeichert.  
Einer mit allen verschiedenen Values  
Einer mit der Startposition der Value

### **cluster encoding:**

Ist gut wenn eine Spalte viele identische Werte hat die hintereinander stehen.  
Attribut Vektor ist partitioniert in n Blöcke mit fester Größe (typischerweise 1024 Elements)  
Wenn ein Cluster nur einen Wert hat wird er durch eine 1 ersetzt.  
Wurde er nicht ersetzt steht dort eine 0.

### **sparse encoding:**

o index encoding:

Ist gut wenn verschiedene Values oft vorkommen  
BSP: bei zusammenhängenden Spalten. Nach Land Sortiert und auf Namensspalte zugreifen  
Wie bei Cluster encoding N Datenblöcke mit fester Anzahl Elementen (1024)

Die SAP Hana Datenbank benutzt Algorithmen um zu entscheiden, welche der Komprimierungsmethoden am angebrachtesten für die verschiedenen Spalten ist.

Bei jeder „delta merge“ Operation wird die Datenkompression automatisch evaluiert, optimiert und ausgeführt.

- In-Memory Datenbank
- Column-Based Architektur
- Komprimierung
- Memory Zugriffe

## 5 Star Schema Benchmark (SSBM)

Der Star Schema Benchmark (SSB) wurde von Pat O’Neil, Betty O’Neil und Quedong Chen entwickelt, um die Performance von Datenbanksystemen, welche mit Data-Marts nach dem Star Schema arbeiten, zu ermitteln und Vergleichbar zu machen [Star Schema Benchmark Quelle]. Dabei nutzen sie das bekannte TPC-H Benchmark [TPCH Quelle] als Grundlage für ihr Star Schema Benchmark, modifizieren es jedoch vielfach zugunsten eines guten Star Schemas.

### TPC-H zu SSB-Transformation

Die von Chen, O’Neil und O’Neil durchgeführten Transformationen von TPC-H zu SSB wurden an die von Kimball und Ross erläuterten Prinzipien zur Dimensionalen Modellierung [**The Data Warehouse Toolkit Second Edition - Quelle einfügen**] angelehnt.

— Hier SSB-M Schema Grafik einfügen —

Im Folgenden sind die wichtigsten Änderungen kurz zusammengefasst:

1. Die beiden Tabellen LINEITEM und ORDER aus dem TPC-H Schema werden in SSB zu einer gemeinsamen Tabelle LINEORDER zusammengefasst, was als Denormalisierung bezeichnet wird [**The Data Warehouse Toolkit Seite 121 - Check**]. Dadurch werden für gängige Abfragen weniger Joins benötigt. Die Kardinalität der Tabelle entspricht der ursprünglichen LINEITEM Tabelle und beinhaltet einen replizierten ORDERKEY zur Verknüpfung der Tabellen.
2. Die Tabelle PARTSUPP aus dem TPC-H Schema wird nicht in das SSB übernommen, da die Granularität zwischen PARTSUPP und LINEORDER nicht übereinstimmt. Dies kommt daher, dass LINEORDER bei jeder Transaktion vergrößert wird, die PARTSUPP Tabelle jedoch nicht. Sie hat lediglich die Granularität Periodic Snapshot, da es keinen Transaction Key für sie gibt. Auch im TPC-H Schema gibt es keine Aktualisierungen über den Verlauf. Damit bleibt sie im Gegensatz zur LINEORDER Tabelle über den Zeitverlauf unverändert.

Dies würde kein Problem darstellen, wenn PARTSUPP und LINEORDER durchgehend als getrennte Faktentabellen behandelt würden, welche nur getrennt abgefragt und nie zusammengefügt werden. Jedoch zeigt Abfrage Q9 aus dem TPC-H Schema, dass LINEITEM, ORDERS und PARTSUPP kombiniert werden, womit Konflikte entstehen.

Die Autoren des SSB-M argumentieren, dass die PARTSUPP Tabelle im Kontext eines Data Marts unnötig ist, woraus die Löschung der Tabelle erfolgt. Stattdessen wird eine Spalte SUPPLYCOST aus der Tabelle zu jeder LINEORDER Zeile im neuen Schema hinzugefügt. Dadurch wird die Korrektheit der Information in Bezug zur Bestellzeit sicher gestellt.

TODO: Für andere Transformationsdetails von TPC-H zu SSB verweisen wir den Leser auf [Star Schema Benchmark]. Beispielsweise werden die Spalten TPC-H SHIPDATE, RECEIPTDATE und RETURNFLAG gelöscht, da die Bestellinformationen vor dem Versand abgefragt werden müssen, und wir wollten uns nicht mit einer Folge von Faktentabellen befassen, wie in [Kimball, Ross], pg. 94. Außerdem hat TPC-H keine Spalten mit relativ kleinem Filterfaktor, daher fügen wir eine Anzahl von Rollup-Spalten hinzu, wie P\_BRAND1 (mit 1000 Werten), S\_CITY und C\_CITY und so weiter.

- Warum SSBM? Für Dimensionale Modellierung, interessant für OLAP
- Unterschiede zu TPC-H ausarbeiten anhand SSB-M Schema, Quellen, Bilder
- Generierung von SSBM-Tabellen
- Tabellen in HANA laden

## 6 Durchführung von Benchmarks

### Aufsetzen von HANA: Installation, Beschreibung vom System (Prozessoren, RAM, OS, Festplattenspeicher etc.)

Für die Durchführung vom Benchmark wurde auf einem Dell Latitude E5570 verwendet.

Die wichtigsten Merkmale:

CPU: Intel i7-6820HQ CPU @ 2.70 GHz (4 Cores, 8 Threads)

RAM: 16GB DDR3 @ 2133Mhz

Storage: USB3.0-SSD

HANA wurde in Form einer Virtuellen Maschine über den HXEDownloader von <http://sap.com/sap-hana-express> bezogen. Die VM gibt es in einer Server only Version und einer Server + Applications Version. Die Tests wurden auf der Server + Applications Version durchgeführt. Um Overhead durch die Virtualisierung zu verhindern, wurde das Festplattenimage der VM auf die SSD extrahiert und das System von dort gebootet. Zum extrahieren wurde quem-img verwendet:

```
1 sudo qemu-img convert -O raw hxexsa-disk1.vmdk /dev/sdb
```

Das Betriebssystem ist SUSE Linux Enterprise Server 12 SP2. Wegen Hardware Kompatibilitätsproblemen wurde der Kernel nachträglich auf 4.4.117-3 aktualisiert.

### Durchführung von Performance Tests

#### Vorbereitung

In der HANA-Datenbank wurde das SSBM-Schema angelegt. Die Tabellen den für SSBM-Benchmark wurden mit Hilfe von SSBM-Tabellengenerator dbgen generiert (mit Scaling Factor 1 für 1GB Daten) (<https://github.com/electrum/ssb-dbgen>).

```
dbgen -s 1 -T a
```

Die generierten CSV-Tabellen wurden dann in die Datenbank geladen.

```
1 IMPORT FROM CSV FILE '/hana/shared/HXE/HDB90/work/date.tbl' INTO "SYSTEM"."DIM_DATE"
```

```
2 WITH
3
4 record delimited by '\n'
5 field delimited by '|';
```

Dieses Vorgehen, die Tabellen komplett mit einem Import-Statement zu laden hat zur Folge, dass bei den Abfragen die gesamten Daten in der Basis-Tabelle waren und die Delta-Tabelle leer war.

## Ladezeiten von Tabellen und Indizes

Bereits beim Laden der Tabellen wurde der Unterschied zwischen Spalten- und Zeilen-basierten Speicherung festgestellt. Der Ladeprozess bei der Spalten-basierten Tabellenorganisation hat 27% weniger Zeit benötigt (81 Sekunden für Column Store und 112 Sekunden für Row Store). Ein möglicher Grund ist die Kompression, die dafür sorgt, dass weniger Daten geschrieben werden müssen.

Als nächstes haben wir die Ladezeiten für das Anlegen der Indizes gemessen. Es wurden Indizes für Spalten mit unterschiedlich vielen einmaligen Werten in unterschiedlich großen Tabellen ausgewählt (*LO\_ORDERKEY* und *LO\_DISCOUNT* auf der Faktentabelle und *D\_YEAR* auf einer Dimensionstabelle).

Bei Spalten-basierten Tabellen war das Anlegen von Indizes um einiges schneller. Der Unterschied war um so größer je weniger verschiedene Werte in der Spalte vorhanden waren (um Faktor 14 bei *LO\_ORDERKEY* und um den Faktor 37 bei *LO\_DISCOUNT*).

Bei *D\_YEAR* war das Erstellen vom Index bei den Zeilen-orientierten Tabellenorganisation schneller. Da das Anlegen von diesem Index jedoch insgesamt sehr schnell war, kann das darauf zurückzuführen sein, dass der Overhead zu groß ist und dass dagegen die eigentliche Zeit zum Erstellen von Indizes verschwindend gering ist. Um eine genauere Aussage treffen zu können, sind weitere Informationen über die internen Datenstrukturen der HANA-Datenbank notwendig, zu denen uns keine Dokumentation vorliegt.

## Vorgehensweise

Das Ziel des Benchmarks war, Star Schema auf HANA-Datenbank zu testen. Der Schwerpunkt lag dabei beim Vergleich zwischen Spalten- und Zeilen-basierten Tabellnorganisation. Es ging vor allem darum, am Beispiel von HANA In-Memory-Datenbank zu testen, ob Columnstore sich besser für Data Warehouse bzw. OLAP-Zwecke eignet als Zeilen-basierte Datenspeicherung. Desweiteren wurde der Einfluss von Indizes auf die Performance von HANA-Datenbank bei Column- und Rowstore analysiert.

Der Benchmark wurde mit folgenden Testvariablen durchgeführt:

- Tabellenorganisation
- Indizes
- Hints

- Anzahl von CPUs.

Die Tests wurden iterativ mit verschiedenen Kombinationen der Testvariablen durchgeführt. Die Durchführung des Benchmarks lässt sich in folgende Schritte unterteilen:

1. Erzeugung vom Schema und Datenimport (Wechsel zwischen Column- und Rowstore)
2. Erstellen von Indizes
3. Durchführung von Benchmarks (jeweils 100 Iterationen):
  - ohne Hints
  - mit Hint `USE_OLAP_PLAN`
  - mit Hint `NO_USE_OLAP_PLAN`
4. Speicherung der Daten in einer Log-Datei
5. Importieren der Daten in den Cube
6. Analyse und Auswertung der Ergebnisse

Um den Einfluss von asynchronen Prozessen auf die Testergebnisse zu vermeiden, wurden die Benchmarks für Row- und Columnstore getrennt durchgeführt. Die Erzeugung vom Column- bzw. Row-Schema und der Datenimport (Schritt 1) erfolgten daher manuell.

Schritte 2-4 wurden automatisiert mit einem bash-Skript ausgeführt. Für die Durchführung des Benchmarks wurden SQL-Abfragen zum Anlegen und Entfernen von Indizes, sowie SSBM-Abfragen (mit und ohne Hints) vorbereitet, die im bash-Skript nacheinander ausgeführt wurden. Benchmarks mit unterschiedlichen Indizes wurden jeweils ohne Hints sowie mit und ohne OLAP-Hint durchgeführt.

Damit der Benchmark zuverlässige Ergebnisse liefert, wurden alle Kombinationen der Testvariablen jeweils 100 mal ausgeführt. Mehrere Iterationen sind hilfreich, um Anomalien und zufällige Einflussfaktoren bei der Durchführung der Tests auszuschließen.

Die Ergebnisse der Tests wurden in eine Log-Datei geschrieben, die mit Hilfe von einem selbsterstellten Java-Programm (BenchmarkLoader) geparkt und in einen virtuellen Cube in die HANA-Datenbank geladen wurden. Der Cube eignet sich gut für die Auswertung der Benchmark-Ergebnisse, da wir unterschiedliche Testvariablen haben, die in verschiedenen Kombinationen getestet werden.

Im Folgenden wird die Auswahl von Indizes, der BenchmarkLoader und der virtuelle Cube beschrieben.

## Auswahl der Indizes

Die Indizes wurden in verschiedene Kategorien eingeordnet. Zunächst wurden Indizes auf die Fremdschlüssel-Spalten in der Faktentabelle angelegt. Danach wurden zusätzliche Indizes auf die Attributen der Faktentabelle hinzugefügt. Indizes auf Primärschlüssel erstellt HANA implizit, deshalb wurden sie nicht explizit getestet [###].

- ```
1 LO_CUSTKEY
2 LO_SUPPKEY
3 LO_PARTKEY
```

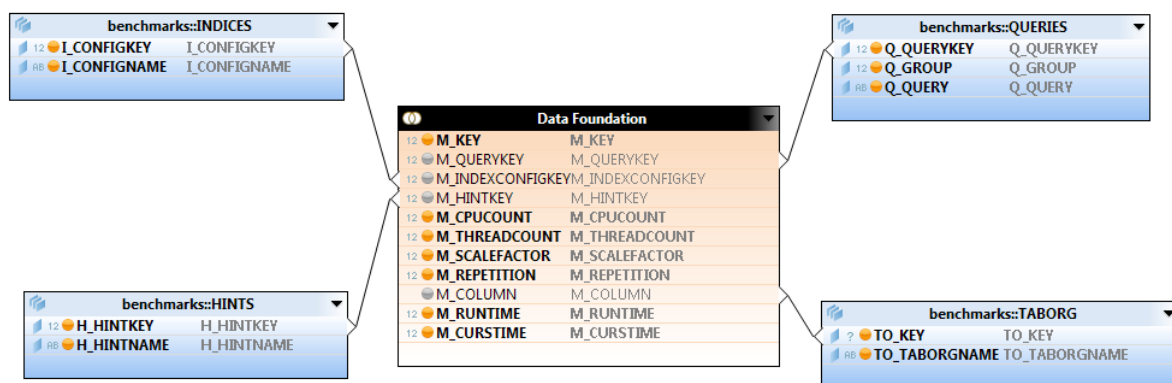
```
4 LO_ORDERDATEKEY
5 LO_COMMITDATEKEY
6
7     +
8     LO_QUANTITY
9     LO_EXTENDEDPRICE
10    LO_DISCOUNT
11
12    +
13    C_REGION
14    C_MRKTSEGMENT
15    P_MFGR
16    P_CATEGORY
17    S_NATION
18    S_REGION
19    D_YEAR
20
21    +
22    C_CITY
23    P_BRAND
24    S_CITY
25    D_YEARMONTHNUM
26    D_YEARMONTH
27    D_DAYNUMINYEAR
28
29 - remove all fact table indices = DimOnly
```

Bei den Dimensionstabellen wurden Indizes auf restriktive und weniger restriktive Spalten getestet. So schränkt beispielsweise eine Bedingung auf die Region kaum ein, weil eine Region sehr groß ist im Vergleich zu einer Stadt, die die Treffermenge stark einschränkt.

## BenchmarkLoader

## Benchmark-Cube

Die Benchmark-Daten wurden in der HANA-Datenbank in einem Star Schema gespeichert. Die Messdaten in der Faktentabelle sind die Ausführungszeiten, die vom Server reportet werden *TOTALTIME* (*RUNTIME* + *CURSTIME*, Runtime ist die Server-Zeit, um die Ergebnisse zu berechnen, und die Curstime - um die Ergebnisse auszuliefern). Die Benchmark-Ergebnisse sind multidimensionale Daten. Jede Testvariable entspricht einer Dimension: Tabellenorganisation (Row- oder Columnstore), SSBM-Queries, Indizes und Hints.



**Abbildung 6.1:** Benchmark-Cube

Man soll jedoch vermeiden, dass der Cube sparse besetzt ist (wenn Daten zu bestimmten Testvariablen fehlen), und möglichst nach verschiedenen Parametern filtern, um keine falschen Schlussfolgerungen zu ziehen. Des Weiteren soll bei der Auswertung der Messungen die Durchschnittszeiten und keine Summe verglichen werden, um zu vermeiden, dass die Tests, die öfter durchgeführt werden, größere Werte liefern (z.B. wenn Columnstore mehr als mit Rowstore getestet wurde).

## Testanalyse Auswertung der Query Execution Plans

Ohne Indizes schneidet der Column Store mit großem Abstand bei jeder SQL-Query besser ab als Row Store. Wenn man die Indizes hinzufügt,

Mit Fremdschlüssel performt Columnstore immer gut, Bei Rowstore ist die Performance stark von den Queries abhängig. Bei manchen Queries performt Rowstore gut, und kommt manchmal an die Performance von Columnstore, aber aber oft wesentlich schlechter.

Auffällig war, dass Rowstore mit Indizes manchmal schlechter performt als ohne Indizes.

Beispiel: Q3.1 und Q3.3

Columnstore profitiert von Indizes, allerdings nicht so stark als Rowstore.

### // TODO

- Q3.1 vs. Q3.3 mit QEP
- Column Store mit Indizes schneller, aber QEPs sind gleich
- HINTs
- CPUs
- Cube
- Excel



## **7 Fazit**

## 8 Anhang

```
1 {"General": {
2     "Repetitions": 100,
3     "ScalingFactor": 1
4 },
5 "column_benchmark_no_index": {
6     "column": true,
7     "index": "none",
8     "hint": "none",
9     "CPU": 4,
10    "Threads": 8,
11    "repetitions": [
12        [
13            {
14                "Type": "exec_file",
15                "Filename": "./sql/benchmark/q1_bench/q1.1.sql",
16                "times": " 13732;"
17            },
18            {
19                "Type": "exec_file",
20                "Filename": "./sql/benchmark/q1_bench/q1.2.sql",
21                "times": " 14713;"
22            },
23            ...
24            {
25                "Type": "exec_file",
26                "Filename": "./sql/benchmark/q4_bench/q4.3.sql",
27                "times": " 20654;333;"
28            }
29        ],
30        [
31            {
32                "Type": "exec_file",
33                "Filename": "./sql/benchmark/q1_bench/q1.sql",
34                "times": " 12546;12788;9118;"
35            },
36            {
37                "Type": "exec_file",
```

```
38         "Filename": "./sql/benchmark/q1_bench/q1.1.sql",
39         "times": " 14242;"
40     }
41     ...
42 ]
43 ]
44 },
45 "column_benchmark_no_index_noolap": {
46     "column": true,
47     "index": "none",
48     "hint": "NO_USE_OLAP_PLAN",
49     "CPU": 4,
50     "Threads": 8,
51     "repetitions": [
52         [
53             ...
54         ]
55     ]
56 }
57 ...
58 }
```