INTRODUCTION TO COLUMN STORES WITH PIVOTAL GREENPLUM

SEMINAR DATABASE SYSTEMS
MASTER OF SCIENCE IN ENGINEERING
MAJOR SOFTWARE AND SYSTEMS
HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
WWW.HSR.CH/MSE

SUPERVISOR: PROF. STEFAN KELLER
AUTHOR: ANDREAS EGLOFF

DAVOS, DECEMBER 2015

ABSTRACT

Today's relational databases order their data in rows, thus enabling fast access for most queries. However, there exists another way of saving the desired data. Instead of storing rows in a traditional row-based manner, the data is stored in columns. This paper outlines how a column-oriented DBMS (column store) is setup on the basis of Pivotal Greenplum. Additionally, the disk usage and the query performance of row stores and column stores are run against each other. Furthermore, the use of run length encoding is incorporated as well, thus reducing the disk usage once more. It is shown that the disk usage can be shrinked dramatically with column stores, whereas the query performance has no noteworthy speed gain with the test data given.

KEYWORDS: COLUMN-ORIENTED DBMS, COLUMN STORE, COLUMNAR COMPRESSION

TABLE OF CONTENTS

1 Introduction

1.1 Overview

Today's relational databases order their data in rows, thus enabling fast access for most queries. However, there exists another way of saving the desired data. Instead of storing rows in a traditional row-based manner, the data is stored in columns. This paper outlines how a column-oriented DBMS (column stores) is setup on the basis of Pivotal Greenplum. Specifically, there will be examples of how a new table is initialized in a columnar order as well as an exemplary transformation of an existing row-based table. Additionally, it will show how this way of fetching data differs and points out pros and cons of each method. In order to support the latter, this paper presents a benchmark that will show in numbers how significant the difference is. Finally, it presents some lessons learned in terms of the findings.

1.2 Column-oriented Databases

A classical relational database is row-oriented. Each row adds new data for the given columns. In a column-oriented database the data is stored by their columns. The only fundamental difference is therefore in the storage layout. This little difference might appear as a minor alteration, however, one needs to look at the bigger picture. For that, we need to delve into what data is retrieved and how. Let us consider the following table layout:

fid	name	class	state	county	ele	map
001	Cypress Slough	Stream	AR	Miller	60	Domino
002	Days Creek	Stream	AR	Miller	54	Doddridge NW
003	McKinney Bayou	Stream	AR	Miller	55	Doddridge NE
004	Nix Creek	Stream	AR	Miller	82	Texarkana
005	Rocky Creek	Stream	AR	Miller	75	Domino
006	Sulphur River	Stream	AR	Miller	56	Doddridge SE
007	Texarkana Post Office	Post Office	AR	Miller	102	Texarkana
008	Old River Lake	Lake	AR	Little River	94	Foreman
009	State Line Creek	Stream	LA	Caddo	56	McLeod

TABLE 1-1

Crucial to the performance of a query execution is how the disc accesses the data and this is reflected by how many rows must be read. In a traditional row-based system executing a query results in accessing the whole row or even a costly full table scan. To overcome this problem, indexes are introduced. However, they come with their own overhead as well and need to be maintained regularly [1]. In our example we want to fetch only the data that is in *Miller* country in the state *AR*. This results in a full table scan which in case of billions of rows leads to slow performance. The retrieval looks like this:

```
001, Cypress Slough, Stream, AR, Miller, 60, Domino,  002, Days Creek,
Stream, AR, Miller, 54, Doddridge NW,  003, McKinney Bayou, Stream, AR,
Miller, 55, Doddridge NE,  004, Nix Creek, Stream, AR, Miller, 82,
Texarkana,  005, Rocky Creek, Stream, AR, Miller, 75, Domino,  006,
Sulphur River, Stream, AR, Miller, 56, Doddridge SE,  007, Texarkana
Post Office, Post Office, AR, Miller, 102, Texarkana,  008, Old River
Lake, Lake, AR, Little River, 94, Foreman,  009, State Line Creek,
Stream, LA, Caddo, 56, McLeod
```

LISTING 1-1

That is where column-oriented database systems, also known as column stores, come in and prove their strength. In column stores the data is organized in columns rather than rows, so instead of entire rows the entire column is read. This can result in significant query efficiency. Projecting this concept to our example we only look at column *state* and *county*. In this case the retrieved records for column state are

```
001 AR, 002 AR, 003 AR, 004 AR, 005 AR, 006 AR, 007 AR, 008 AR, 009 LA,
001 Miller, 002 Miller, 003 Miller, 004 Miller, 005 Miller, 006 Miller,
007 Miller, 008 Little River, 009 Caddo
```

LISTING 1-2

where the numbers represent the row numbers or *fid*. This step is required for assigning the corresponding data. Now, we can easily see that only the data from the first row up to the seventh row is relevant. Moreover, with compression algorithms the data to be retrieved can be optimized drastically as it repeats itself.

A handful of commercial DBMS exist and often offer both, row- and column-oriented column stores. The most popular ones are HBase, Cassandra, Hypertable, CStore, MonetDB just to name a few. In this paper we highlight Pivotal Greenplum, which comes with a columnar option as well.

1.3 What are the Benefits?

There are several selling points why one wants to work with column stores. Being developed with *Big Data* in mind, they make use of horizontal scalability, the ability to connect multiple hardware (or software) entities that work as a single logical unit. Horizontal scaling itself, also known as scaling out, allows to increase the capacity on the fly.

Apart from the obvious, namely the faster querying (especially for ad-hoc queries) of high volume and read-intensive data, adding columns to column stores is inexpensive and there is no real storage cost for unpopulated values. As a consequence, calculating an aggregate over many rows happens faster as well. This is why column stores are well suited for OLAP environments which typically involve a small number of highly complex queries over terabytes of data [2].

In column stores there is almost always support for features like versioning and compression. In sub-chapter 1.5 there will be a short abridgment about compression in column stores.

Column-oriented organizations are more efficient when new values of a column are supplied for all rows at once, because that column data can be written efficiently and replace old column data without touching any other columns for the rows [3].

Finally, the decisive point when comparing column stores with traditional row stores is the efficiency of the hard disc access for a given query. As a hard disk's seek time is today's bottleneck compared to the much more improving CPU performance, the way the data is accessed (when not in-memory) can be a game changer. This is especially the case when querying tables with hundreds of columns but only a few columns are relevant, thus pointing out a notable difference in access time as we try to prove in chapter 3.

1.4 What are the disadvantages?

So why is not everyone using column stores instead of the "old" row stores? After all the convincing facts, we will talk about the drawbacks that such a system brings along.

First of all, column stores restrict the database schema design. That is why many allow a mixture between row and column-based database system to have the best of both worlds. An example is Google's row-based BigTable which features a "column group" to avoid frequently needed joins [4].

Second of all, row stores are more efficient when many columns of a single row are required at the same time as well as when the row size is relatively small, thus enabling data retrieval with a single disk seek. The same applies to writing data, when all the row data is supplied at the same time.

As a result, row stores are more efficient with OLTP transactions, whereas column stores go strong with OLAP workloads such as in a data warehouse.

Finally, all the different database patterns like relational, key-value, columnar, document and graph (some of which are not described here) have their advantages and drawbacks. It turns out, the best practice is to choose your pattern based on how you plan to query the data not just what it consists of.

1.5 Compression

The goal for ordering data in a columnar way is structuring similar data and consequently make use of compression techniques that are not available in row stores. Popular techniques include LZW and run-length encoding (RLE), only one of which is described in this paper.

As a result, when organizing the data in a column store, the disk space needed is much less. However, As the data now must be uncompressed to be read which on its part costs precious CPU time, various mechanisms are apparently in place to minimize the need for access to compressed data [5].

2 Columnar Data in Pivotal Greenplum

2.1 Environment

Greenplum Database by the company Pivotal is a massively parallel processing (MPP) database server based on PostgreSQL. This means that more than one processor is involved to carry out an operation. In order to manage the operation requests there exists a so-called master which acts as the entry point to the database. The master coordinates the workload across all database instances connected to the master, so-called segments, which are ultimately responsible for the data processing as well as storage. The segments themselves communicate with each other and the master over a networking layer of the Greenplum database, the so-called interconnect. Figure 2-1 depicts the essence of this setup.

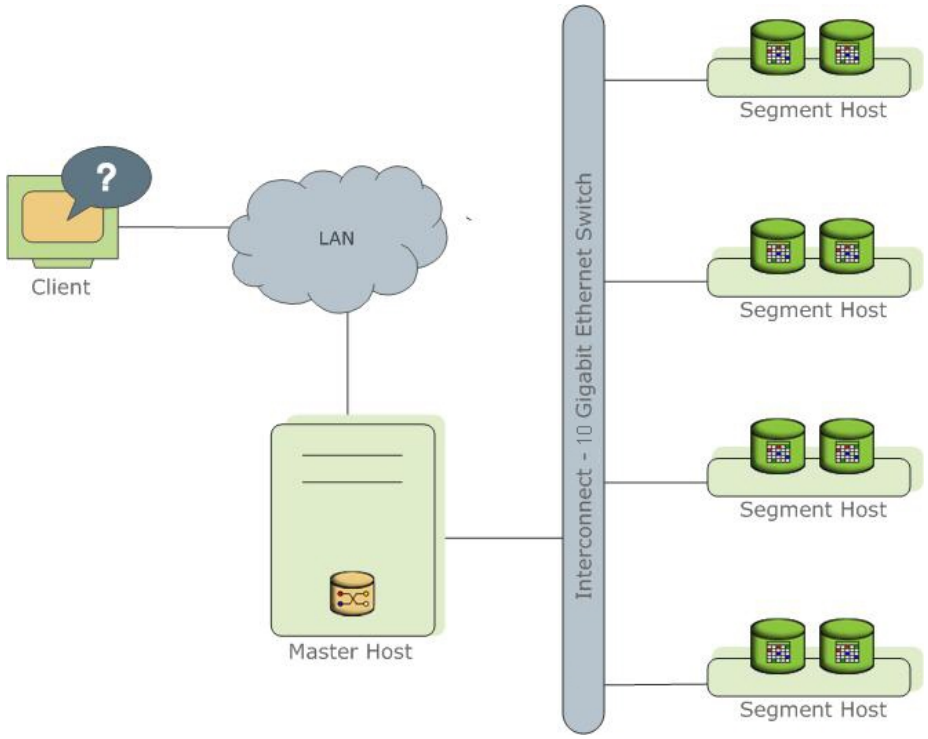

FIGURE 2-1

Because of its distributed nature across multiple machines, a proper selection and configuration of hardware is vital to achieving best possible performance. The following subsections will feature a quick glimpse into the components from above.

2.1.1 The Greenplum Master

As described above, the master is the entry point to the database system. As such, it plays a vital role in keeping the database system together. A user can connect to the master using a PostgreSQL-compatible client such as psql or any JDBC driver for that matter. The user only sees the master as such and not any segment host behind. The master itself does not store any user data, it just authenticates the client connections, processes incoming SQL commands, distributes the workload between the segments and finally presents the results to the client.

Because of its critical role in the system, there exists an option for a master redundancy. This warm standby kicks in the moment the master becomes nonoperational and can be deployed on a separate machine or on one of the segment hosts.

All in all, it is wise to offer the master a fast, dedicated CPU for data loading, connection handling and query planning.

2.1.2 The Segments

As already stated, a user does not directly interact with the segment but rather with the master. However, the data itself is stored on the segments, even more, the data is distributed across the segments, so one segment only holds a distinct portion of the whole.

As seen with the master, there exists an option for segment redundancy as well. In order to be able to do that there must be at least more than one segment host, thus the mirror segment always resides on a different host than its primary segment.

It has proven to be reasonable to offer each segment instance on a segment host a single CPU or CPU Core. So if we have a segment host featuring three quad-core processors we may have three, six or twelve segments on this segment host.

2.1.3 *The Interconnect*

The Interconnect is the networking layer of the Greenplum database. It represents the inter-process communication between the segments and uses a standard 10 Gigabit Ethernet switching fabric. For sending messages over the network it uses UDP by default.

2.2 Table Storage Models

In a Greenplum database several storage models are supported, some of which will be presented here. The process of choosing a storage model happens right before table creation as it cannot be changed afterwards.

2.2.1 *Heap Storage*

This is the default storage model as seen in PostgreSQL. It works best in an OLTP environment with lots of updates after initial loading.

2.2.2 *Append-optimized Storage*

This type of storage model works best with denormalized fact tables in a data warehouse environment. When working with large tables, changing the storage layout to append-optimized storage will eliminate the storage overhead and save about 20 bytes per row. It is not recommended for single row INSERT statements, instead bulk loading is greatly optimized.

A table can be created as an append-optimized storage with a respective WITH clause:

```
CREATE TABLE bar (a int, b text)
  WITH (appendonly=true)
  DISTRIBUTED BY (a);
```

LISTING 2-1

2.2.3 *Column-oriented Storage*

As described in detail in chapter 1, column stores are also supported in Greenplum. As also stated they play nicely in data warehouse environments where aggregations are computed over a small number of columns.

As for data types, there is no restriction as to what types have to be used or are supported in column stores [6].

A column-oriented table can be created with a respective WITH clause:

```
CREATE TABLE bar (a int, b text)
  WITH (appendonly=true, orientation=column)
  DISTRIBUTED BY (a);
```

LISTING 2-2

Note that column-oriented tables are only available with the append-only option. As a table's orientation cannot be changed later, a new table has to be created with the desired orientation. Hence, transforming a row store to a column store only works like this:

```
CREATE TABLE cs (LIKE rs)
  WITH (appendonly=true, orientation=column, compresstype=none);
  INSERT INTO cs SELECT * rs;
```

LISTING 2-3

2.2.4 Enabling Compression

Compression can reduce the size of the data dramatically and may be used on append-only tables. There are two types available:

- Table-level compression, where the whole table gets compressed. The supported algorithms are ZLIB and QUICKLZ.
- Column-level compression, applied to a specific column. Different compression algorithms within the same table are possible. The supported algorithms are RLE_TYPE, ZLIB and QUICKLZ.

When using compression one should keep in mind that it is very much depending on the hardware environment on which the segment is running. QUICKLZ compression uses less CPU power while compressing data faster at a compression ratio lower than ZLIB. On the other hand, ZLIB provides higher compression ratios at lower speeds. It is recommended to perform comparison tests to find out the right settings on the specific machine.

A compressed table can be created with a respective WITH clause:

```
CREATE TABLE foo (a int, b text)
  WITH (appendonly=true, compresstype=zlib, compresslevel=5);
```

LISTING 2-4

It is also possible to add different compressions to single columns:

```
ALTER TABLE T1
  ADD COLUMN c4 int DEFAULT 0
  ENCODING (COMPRESSTYPE=zlib);
```

LISTING 2-5

In Greenplum, run-length encoding support is available only for column stores. It works best with loads of repeated data within a column. In contrast, it is not recommended with files that do not have large sets of repeated data as it can greatly increase the compression speed. There are four levels of RLE compression, 1 being the one with the lowest compression ratio but fastest compression speed, 4 being the one with the highest compression ratio but slowest compression speed [7].

2.3 QUERYING COLUMNAR TABLES

There is no special syntax to be used when querying data stored in a column store. So one can use his or her queries used for row stores. The syntax only differentiates when choosing the table storage layout. At this point we present general cases in which it makes sense to use a column store instead of a row store.

- Only a small number of columns from a big table are requested
- Many values of a single column are aggregated and the WHERE or HAVING predicate is also on the aggregate column.
- The WHERE predicate is on a single column and returns a relatively small number of rows.

2.4 Indexes in Greenplum

In order to improve the speed of data retrieval, in most databases indexes are introduced. In Greenplum, however, indexes should be used more sparingly, as it is a distributed database. The use of indexes is not recommended when query workloads generally return very large data sets, as it is not efficient and add significant database overhead. Furthermore, unique indexes are not supported on append-optimized tables.

3 BENCHMARK

3.1 TEST ENVIRONMENT

For testing the performance of the row store, a single node GPDB cluster with 40 TB of space is provided. The installed instance has version 4.3.6.1.

Since the test environment is remote, we open an ssh tunnel like this:

```
ssh -L 63333:localhost:5432 gpadmin@52.23.228.181
```

Now we can run SQL scripts via psql like this:

```
psql -h localhost -p 63333 workbench -U gpadmin -f script.sql
```

3.2 TEST DATA

The test data provided looks as follows:

```
 schemaname | tablename      | no. of tuples
 -------------------------------------------
 public     | osm_poi_tag_ch | 8925668
 public     | osm_poi_ch     | 3324088
 public     | gnis           | 103413
```

LISTING 3-1

The table *gnis* stores geo-referenced places like lakes and streams in the US. *osm_poi_ch* is a collection of coordinates and ids. Combined with *osm_poi_tag_ch* one can get meta-information of these places. Furthermore, there are separate tables for *osm_poi_ch* with 1, 2 and 3 million records.

Because the test system is remote we cannot make a direct comparison with a Postgres instance as desired. That is why we will compare the performance of row stores with column stores within Greenplum and the test data given. So for each data set we have two tables with the same data, one as a row store the other one as a column store.

In order to create a column-oriented table we must first define a distribution key. To ensure even distribution, it is recommended to choose a distribution key that is

unique for each record, preferably the primary key [8]. That is exactly what we choose from now on. Also, the table must be append-only.

Furthermore, we need to choose if we should use compression and if so which algorithm. We choose to use RLE as it is only available on a column level. Now we can define the query as follows:

```
CREATE TABLE gnis_column (
  x double precision not null,
  y double precision not null,
  fid integer not null ,
  name text,
  class text,
  state text,
  county text,
  elevation integer,
  map text
) WITH (appendonly=true, orientation=column, compresstype=rle_type,
compresslevel=4)
    DISTRIBUTED BY (fid);
```

LISTING 3-2

Doing that for all the tables in the test data set, we get the following:

```
                          List of relations
 Schema |          Name          | Type  |  Owner   |       Storage
--------+------------------------+-------+----------+----------------------
 public | gnis                   | table | gpadmin | append only
 public | gnis_column            | table | gpadmin | append only columnar
 public | osm_poi_ch             | table | gpadmin | heap
 public | osm_poi_ch_1mio        | table | gpadmin | heap
 public | osm_poi_ch_1mio_column | table | gpadmin | append only columnar
 public | osm_poi_ch_2mio        | table | gpadmin | heap
 public | osm_poi_ch_2mio_column | table | gpadmin | append only columnar
 public | osm_poi_ch_3mio        | table | gpadmin | heap
 public | osm_poi_ch_3mio_column | table | gpadmin | append only columnar
 public | osm_poi_ch_column      | table | gpadmin | append only columnar
 public | osm_poi_tag_ch         | table | gpadmin | heap
 public | osm_poi_tag_ch_column  | table | gpadmin | append only columnar
```

LISTING 3-3

The disk space used for a table can be dramatically reduced when comparing a table with its columnar counterpart. We execute the following command to get the file size:

```
SELECT pg_size_pretty(pg_total_relation_size('"public"."osm_poi_ch"'));
```

LISTING 3-4

For all tables we get the following sizes:

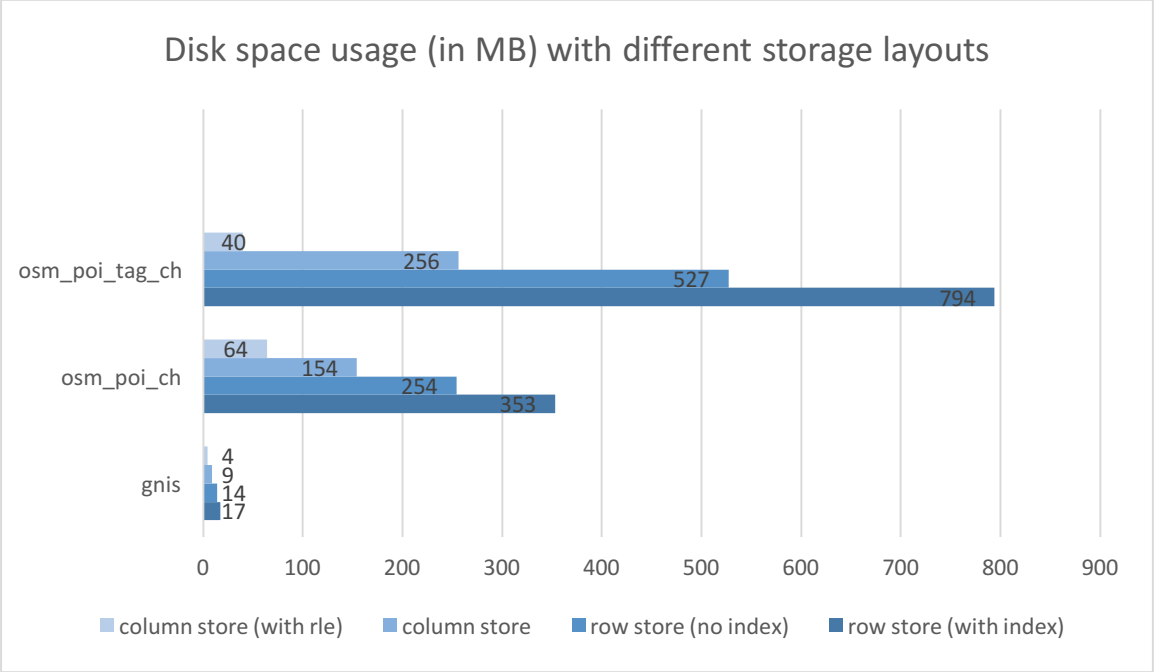

FIGURE 3-1

Note that the compression ratio is much better in *osm_poi_tag_ch* compared to *osm_poi_ch* as there are lots of reoccurring values which enhances the effect of run length encoding. As for columnar storage layouts without any compression, the potential for storage saving is about 50% compared to a row store without indexes . Finally, we can see that the decision of which storage layout to be used greatly depends on the data itself. In the case of the test data, it obviously makes sense using a column store in terms of the disk usage.

3.3 Test Queries

3.3.1 Initialization

There exists a script for measuring how long it takes for creating the 1, 2 and 3 million tables and their indexes. However, a direct comparison between a row and columnar store's creation time is pointless as clustering on append-only tables is not supported [9].

3.3.2 Complex queries

In order to test the performance of more complex queries, we will run the queries from *Appendix A – SQL Scripts* on a row store, a row store with a clustered index, a

column store and a column store with run length encoding. We run the benchmark three times and take the average time. Figure 3-2 and Figure 3-3 depict the results:

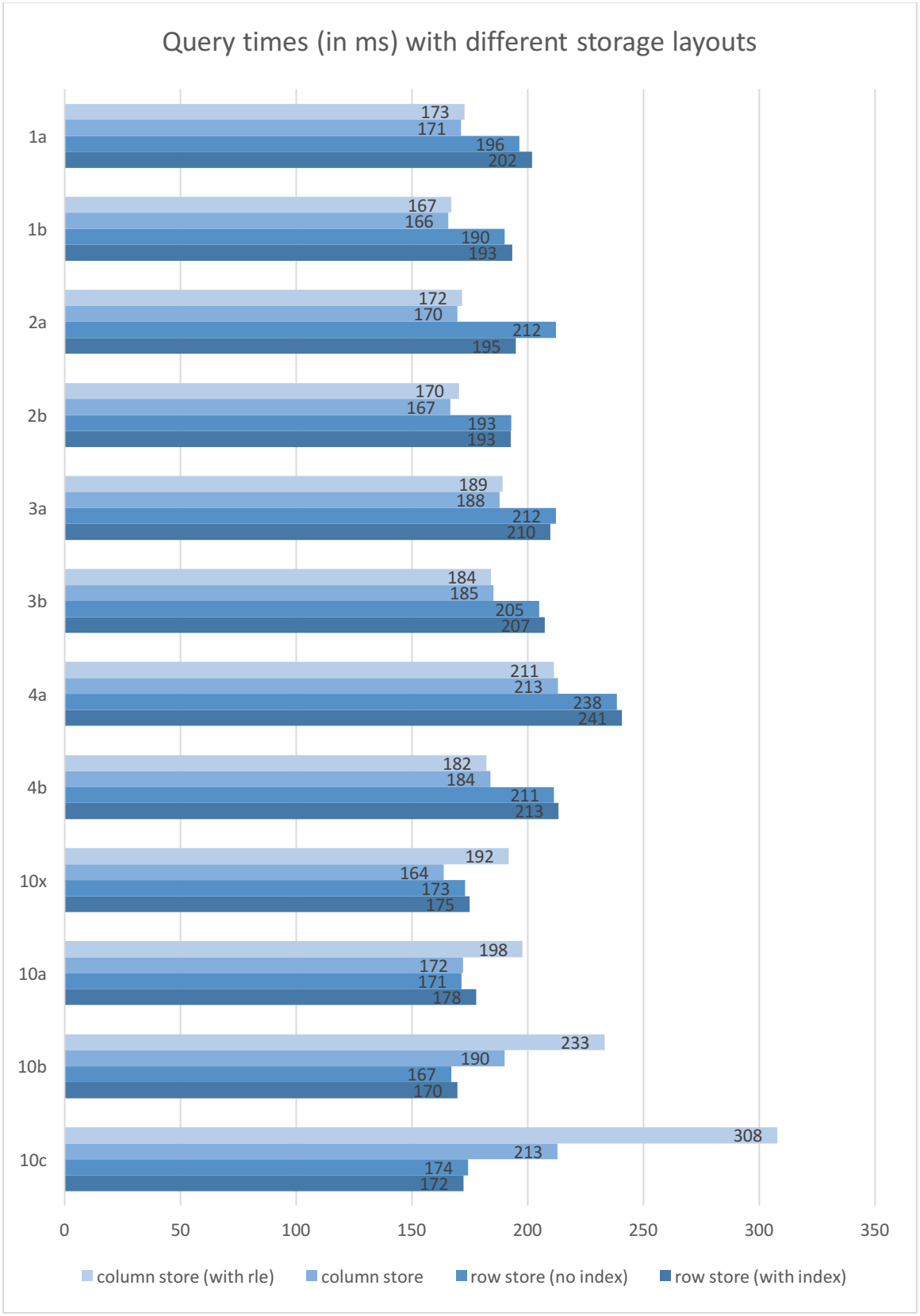

FIGURE 3-2

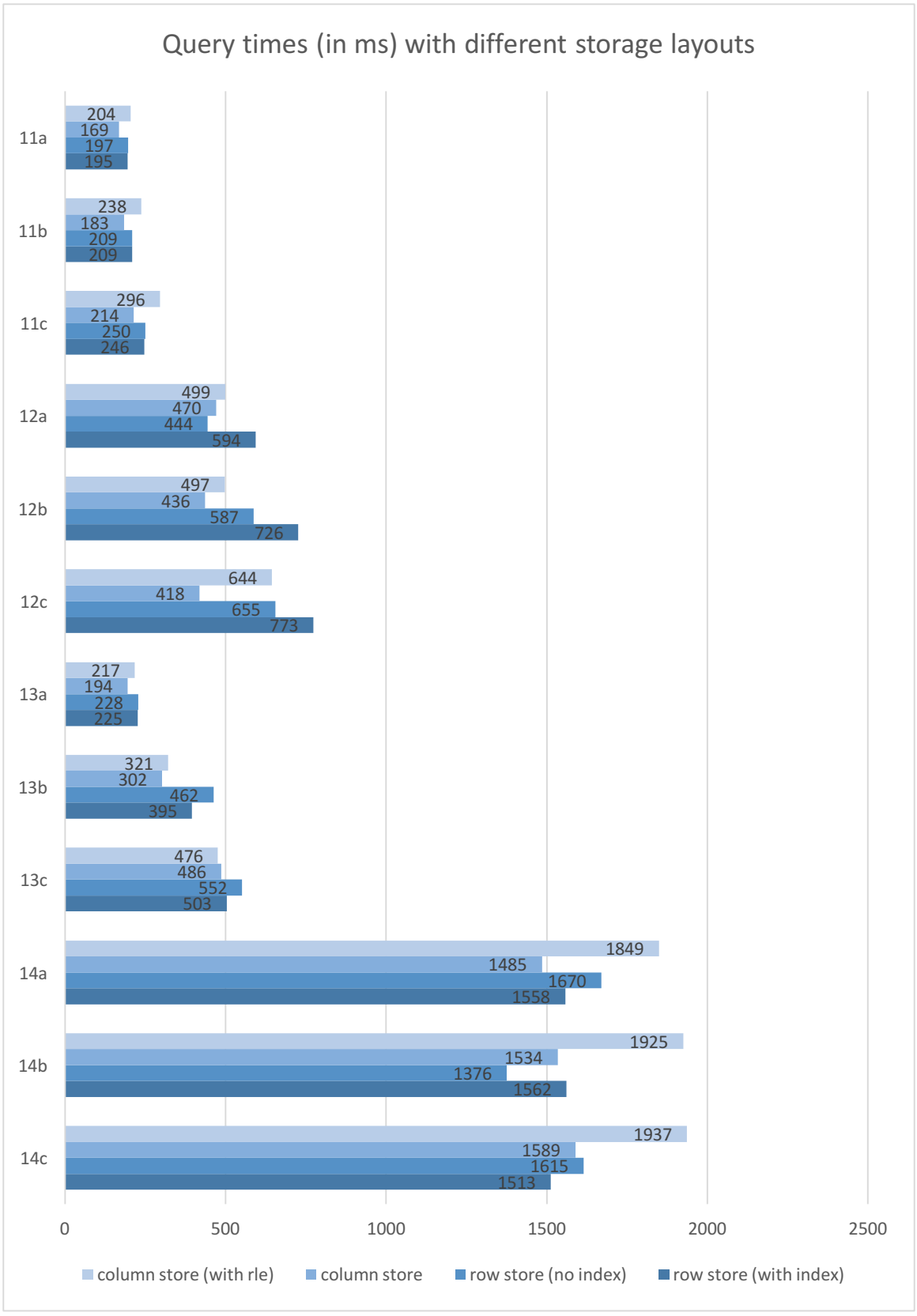
Query times (in ms) with different storage layouts
11a
204
169
197
195
11b
238
183
209
209
11c
296
214
250
246
12a
499
470
444
594
12b
497
436
587
726
12c
644
418
655
773
13a
217
194
228
225
13b
321
302
462
395
13c
476
486
552
503
14a
1849
1485
1670
1558
14b
1925
1534
1376
1562
14c
1937
1589
1615
1513
0
500
1000
1500
2000
2500
column store (with rle)
column store
row store (no index)
row store (with index)

FIGURE 3-3

There are a few things to observe. First, the scripts in set 14 have three joins which make it obviously slower for storage layouts with run length encoding as the results first needs to get uncompressed. Second, and probably most surprising, there is almost always no performance gain when using column stores in favor of row stores. Probably the biggest impact on query speed we can see in 12c where a geographical range condition is requested. However, the performance gain is minor and measurement variations have to be taken into account.

For getting a clearer speed bump we follow best practices. Column stores are best suited for queries that aggregate many values of a single column where the WHERE or HAVING predicate is also on the aggregate column. So the following query falls into this category:

```
SELECT AVG(lon)
FROM osm_poi_ch_3mio
WHERE lon>47.22088 AND lon<47.22974;
```

LISTING 3-5

The performance measurements for this query look like this:

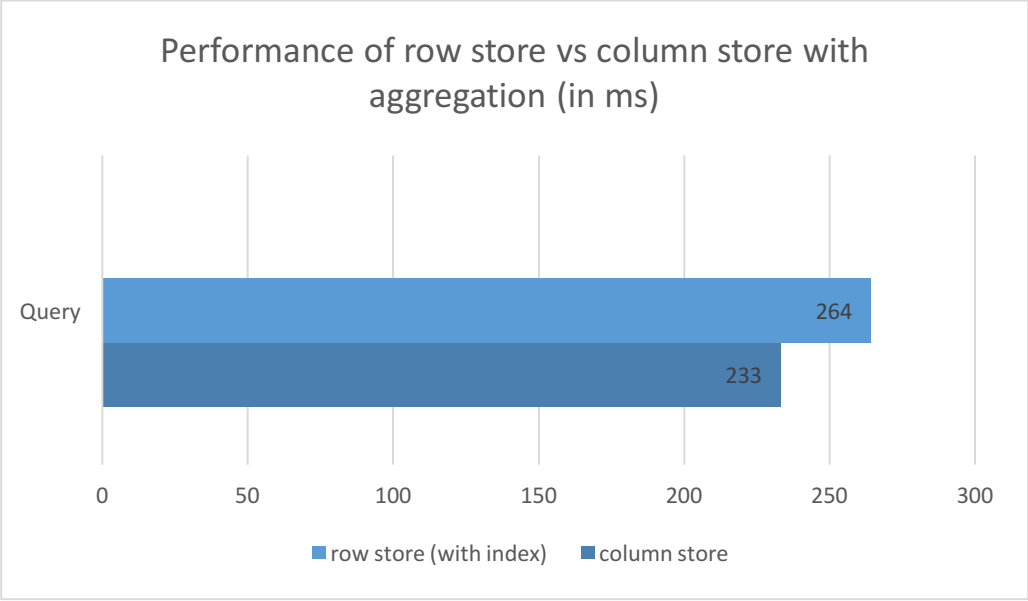

FIGURE 3-4

The performance gain here is once again minor and cannot be taken as a remarkable improvement. Even though the query time is only slightly faster, the table space needed for the column store is 43% of the row store which is quite remarkable given the better performance.

4 Conclusion

First and foremost, we could not make a direct comparison between a standard Postgres instance and a Greenplum instance on the same test machine as planned. This circumstance resulted in the comparison of different storage layouts on a Greenplum instance.

Second, it is shown that with the test data given there were almost no speed improvements when comparing query execution times. However, while no significant improvement in query times could be achieved, the disk usage could be shrinked dramatically.

This finding shows that column stores are not always suited best for organizing data in a database. Maybe with different test data one could achieve different results, as the performance of columnar stores greatly depend on the data itself. For instance, by using bigger tables with thousands of columns. Also, the test data is less than a gigabyte of size which could be too small for finding a remarkable difference in performance, especially when compared to data warehouses with several terabytes of data. The fact that the test system features an insane number of CPU's (36!) could also have its effect. There may also be certain database optimizations in place which are not apparent for the user.

Additional investigations on the test machine have shown, that the data is spread over up to eight segments. However, making use of EXPLAIN ANALYZE does not show if the segments are spread over multiple segment hosts and therefore responsible for possible speed loss. Further work in this area could shed some light on the test results obtained.

As mentioned, the use case for column stores is a data warehouse with a lot of OLAP transactions, for instance with historical data analysis. The canonical example of a good columnar data storage problem is indexing web pages, as these pages are highly textual (benefits from compression) and change over time (benefits from versioning) [10]. Repeating the measurements with such data would be highly interesting but could not be realized within the scope of this work. In spite of everything, one should

note that almost the same query times were achieved while using a columnar table that uses only a fraction of disk space than its row-based counterpart.

Finally, the unbiased attitude when starting these tests is a big lesson learned within this paper, as one could expect a much bigger impact on query performance. As nicely seen, columnar stores greatly depend on the data used, the hardware in use and last and for sure not least, the environment and requirements in general.

5 GLOSSARY

OLTP	On line transaction processing, or OLTP, is a class of information systems that facilitate and manage transaction-oriented applications, typically for data entry and retrieval transaction processing [11].
OLAP	OLAP is an acronym for Online Analytical Processing. OLAP performs multidimensional analysis of business data and provides the capability for complex calculations, trend analysis, and sophisticated data modeling [12].
MPP	MPP (massively parallel processing) is the coordinated processing of a program by multiple processors that work on different parts of the program, with each processor using its own operating system and memory [13].
Horizontal Scalability	Horizontal scalability is the ability to increase capacity by connecting multiple hardware or software entities so that they work as a single logical unit [14].

6 BIBLIOGRAPHY

[1] Sunila Gollapudi, *Getting Started with Greenplum for Big Data Analytics*., 2013, p. 35.

[2] Eric Redmond and Jim R. Wilson, *Seven Databases in Seven Weeks*., 2012, p. 5.

[3] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Column-oriented_DBMS#Benefits

[4] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Column-oriented_DBMS

[5] Dominik Slezak, "Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries," 2008.

[6] Pivotal Greenplum®, *Database Administrator Guide*., 2015, p. 40.

[7] Pivotal Greenplum®, *Database Administrator Guide*., 2015, p. 44.

[8] Sunila Gollapudi, *Getting Started with Greenplum for Big Data Analytics*., 2013, p. 52.

[9] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Column-oriented_DBMS

[10] Eric Redmond and Jim R. Wilson, *Seven Databases in Seven Weeks*., 2012, p. 309.

[11] Wikipedia. [Online]. https://en.wikipedia.org/wiki/Online_transaction_processing

[12] olap.com. [Online]. http://olap.com/olap-definition/

[13] techtarget.com. [Online]. http://whatis.techtarget.com/definition/MPP-massively-parallel-processing

[14] techtarget.com. [Online]. http://searchcio.techtarget.com/definition/horizontal-scalability

7 FIGURES

8 TABLES & LISTINGS

9 APPENDIX A – SQL SCRIPTS

```
-- osm_poi_ch_load.sql
-- Tested on PostgreSQL 9.4 using psql
-- 2015-10-16 SK
-- 2015-11-08 AE column tables

\echo 'osm_poi_ch_loader'


-- Create new table
DROP TABLE IF EXISTS osm_poi_ch_column;

CREATE TABLE osm_poi_ch_column (
  id character varying(64) not null, -- no primary key since OSM id
maybe not unique
  lastchanged character varying(35),
  changeset integer,
  version integer,
  uid integer,
  lon double precision not null,
  lat double precision not null
) WITH (appendonly=true, orientation=column, compresstype=rle_type,
compresslevel=4)
    DISTRIBUTED BY (id);


-- Copy data from CSV file to database
\COPY osm_poi_ch_column FROM 'osm_poi_ch.csv' DELIMITER ';' QUOTE '"'
CSV HEADER;

select count(*) from osm_poi_ch_column;



-- Create new table

DROP TABLE IF EXISTS osm_poi_tag_ch_column;

CREATE TABLE osm_poi_tag_ch_column (
  id character varying(64) not null,
  key text not null,
  value text
  -- primary key (id, key). It is not really true. A dupication found.
)WITH (appendonly=true, orientation=column, compresstype=rle_type,
compresslevel=4)
    DISTRIBUTED BY (id);

-- Copy data from CSV file to the temporary table
\COPY osm_poi_tag_ch_column FROM 'osm_poi_tag_ch.csv' DELIMITER ';'
QUOTE '"' CSV HEADER;

select count(*) from osm_poi_tag_ch_column;
```

```
-- bm_prepare_column_copy.sql

DROP TABLE IF EXISTS osm_poi_ch_1mio_column CASCADE;

CREATE TABLE osm_poi_ch_1mio_column (LIKE osm_poi_ch_1mio)
   WITH (appendonly=true, orientation=column, compresstype=none);
   INSERT INTO osm_poi_ch_1mio_column SELECT * FROM osm_poi_ch_1mio;


DROP TABLE IF EXISTS osm_poi_ch_2mio_column CASCADE;

CREATE TABLE osm_poi_ch_2mio_column (LIKE osm_poi_ch_2mio)
   WITH (appendonly=true, orientation=column, compresstype=none);
   INSERT INTO osm_poi_ch_2mio_column SELECT * FROM osm_poi_ch_2mio;


DROP TABLE IF EXISTS osm_poi_ch_3mio_column CASCADE;

CREATE TABLE osm_poi_ch_3mio_column (LIKE osm_poi_ch_3mio)
   WITH (appendonly=true, orientation=column, compresstype=none);
   INSERT INTO osm_poi_ch_3mio_column SELECT * FROM osm_poi_ch_3mio;
```

```
-- gnis_load.sql
-- Tested on PostgreSQL 9.4 using psql.
-- 2015-10-16 SK
-- 2015-12-20 AE added column orientation, compression none

-- Create table gnis_column:
DROP TABLE IF EXISTS gnis_column;

CREATE TABLE gnis_column (
  x double precision not null,
  y double precision not null,
  fid integer not null ,
  name text,
  class text,
  state text,
  county text,
  elevation integer,
  map text
) WITH (appendonly=true, orientation=column, compresstype=none)
    DISTRIBUTED BY (fid);

-- Copy data from CSV file to database:
\COPY gnis_column FROM 'gnis_names09.csv' DELIMITER ';' QUOTE '"' CSV
HEADER;
```

```
-- Benchmark
-- Tested on PostgreSQL 9.4 using psql
-- 2015-10-16 SK
-- 2015-12-01 AE compatible with Greenplum
--
-- Requirements:
-- * Tables gnis, osm_poi_ch and osm_poi_tag_ch exist and are loaded.

\echo 'Preparing tables. Pls. wait...'

\timing on

\echo '\n=== Table gnis'
-- Preparing index:
DROP INDEX IF EXISTS gnis_fid_idx CASCADE;
CREATE INDEX gnis_fid_idx ON gnis(fid);
CLUSTER gnis_fid_idx ON gnis;                    --USING is not
supported
-- Refreshing statistics:
VACUUM FULL ANALYZE gnis;


\echo '\n=== Table osm_poi_ch'
DROP INDEX IF EXISTS osm_poi_ch_id_idx CASCADE;
CREATE INDEX osm_poi_ch_id_idx ON osm_poi_ch(id);
CLUSTER osm_poi_ch_id_idx ON osm_poi_ch;          --USING is not
supported
VACUUM FULL ANALYZE osm_poi_ch;

\echo '\n=== Table osm_poi_tag_ch'
DROP INDEX IF EXISTS osm_poi_tag_ch_id_idx;
CREATE INDEX osm_poi_tag_ch_id_idx ON osm_poi_tag_ch(id); --103sec
CLUSTER osm_poi_tag_ch_id_idx ON osm_poi_tag_ch;  --USING is not
supported
VACUUM FULL ANALYZE osm_poi_tag_ch;


\echo '\n=== Table osm_poi_ch_3mio'
DROP TABLE IF EXISTS osm_poi_ch_3mio CASCADE;
CREATE TABLE osm_poi_ch_3mio AS
  select
    id,
    max(version) "version",
    max(lastchanged) lastchanged,
    max(uid) uid,
    max(changeset) changeset,
    max(lon) lon,
    max(lat) lat
  from osm_poi_ch
  group by id
  ORDER BY 1 LIMIT 3000000
  DISTRIBUTED BY (id);
ALTER TABLE osm_poi_ch_3mio ADD CONSTRAINT osm_poi_ch_3mio_pk PRIMARY
KEY(id); -- 47sec
CREATE INDEX osm_poi_ch_3mio_pk_idx ON osm_poi_ch_3mio(id); -- 38sec
CLUSTER osm_poi_ch_3mio_pk_idx ON osm_poi_ch_3mio; -- 112sec --USING is
not supported
VACUUM FULL ANALYZE osm_poi_ch_3mio;
```

```
\echo '\n=== Table osm_poi_ch_2mio'
DROP TABLE IF EXISTS osm_poi_ch_2mio CASCADE;
CREATE TABLE osm_poi_ch_2mio AS
  SELECT * FROM osm_poi_ch_3mio
  ORDER BY 1 LIMIT 2000000
  DISTRIBUTED BY (id);
ALTER TABLE osm_poi_ch_2mio ADD CONSTRAINT osm_poi_ch_2mio_pk PRIMARY
KEY(id);
CREATE INDEX osm_poi_ch_2mio_pk_idx ON osm_poi_ch_2mio(id);
CLUSTER osm_poi_ch_2mio_pk_idx ON osm_poi_ch_2mio;  --USING is not
supported
VACUUM FULL ANALYZE osm_poi_ch_2mio;


\echo '\n=== Table osm_poi_ch_1mio'
DROP TABLE IF EXISTS osm_poi_ch_1mio CASCADE;
CREATE TABLE osm_poi_ch_1mio AS
  SELECT * FROM osm_poi_ch_3mio
  ORDER BY 1 LIMIT 1000000
  DISTRIBUTED BY (id);
ALTER TABLE osm_poi_ch_1mio ADD CONSTRAINT osm_poi_ch_1mio_pk PRIMARY
KEY(id);
CREATE INDEX osm_poi_ch_1mio_pk_idx ON osm_poi_ch_1mio(id);
CLUSTER osm_poi_ch_1mio_pk_idx ON osm_poi_ch_1mio;  --USING is not
supported
VACUUM FULL ANALYZE osm_poi_ch_1mio;

\echo '\nOk.'
```

```
-- Benchmark
-- Tested on PostgreSQL 9.4 using psql
-- 2015-10-16 SK
-- 2015-12-02 AE changed to column store

-- Local configuration
\pset format
\pset pager off

-- Redirect query output to file
\set OUTFILE bm_out.txt
\o :OUTFILE

-- This is a dummy query to fill cache with (other) tuples
-- SELECT count(*) FROM osm_poi_tag_ch_column;

\echo '\n=== Table gnis_column'

-- Simple equality search with single tupel in return set
\timing off
SELECT count(*) FROM osm_poi_tag_ch_column;
\timing on
\echo ';1a'
SELECT name, county, state FROM gnis_column t WHERE t.fid=1091310;
\echo ';1b'
SELECT name, county, state FROM gnis_column t WHERE t.fid=1091310;

-- Simple equality search on county Texas:
\timing off
SELECT count(*) FROM osm_poi_tag_ch_column;
\timing on
\echo ';2a'
SELECT name, county, state FROM gnis_column t WHERE t.county='Texas';
\echo ';2b'
SELECT name, county, state FROM gnis_column t WHERE t.county='Texas';

-- Range search with aggregate function
\timing off
SELECT count(*) FROM osm_poi_tag_ch_column;
\timing on
\echo ';3a'
SELECT avg(t.elevation)::int FROM gnis_column t
WHERE t.x>-103.208 and t.y>28.435 and t.x<-96.891 and t.y<33.460;
\echo ';3b'
SELECT avg(t.elevation)::int FROM gnis_column t
WHERE t.x>-103.208 and t.y>28.435 and t.x<-96.891 and t.y<33.460;

-- Group by query
\timing off
SELECT count(*) FROM osm_poi_tag_ch_column;
\timing on
\echo ';4a'
SELECT count(*), class FROM gnis_column GROUP BY class
ORDER BY 1 DESC;
\echo ';4b'
SELECT count(*), class FROM gnis_column GROUP BY class
ORDER BY 1 DESC;
```

```
\echo '\n=== Table osm_poi_ch'

-- Query with equality condition
\timing off
SELECT count(*) FROM gnis_column;
\timing on
\echo ';10x'
select id,version,lon,lat from osm_poi_ch_1mio_column  where
id='1484188127pt';
\echo ';10a'
select id,version,lon,lat from osm_poi_ch_1mio_column  where
id='1484188127pt';
\echo ';10b'
select id,version,lon,lat from osm_poi_ch_2mio_column  where
id='1484188127pt';
\echo ';10c'
select id,version,lon,lat from osm_poi_ch_3mio_column  where
id='1484188127pt';

-- Query with range condition
\timing off
SELECT count(*) FROM gnis_column;
\timing on
\echo ';11a'
select id,version,lon,lat from osm_poi_ch_1mio_column where version>300
order by version desc
limit 10;
\echo ';11b'
select id,version,lon,lat from osm_poi_ch_2mio_column where version>300
order by version desc
limit 10;
\echo ';11c'
select id,version,lon,lat from osm_poi_ch_3mio_column where version>300
order by version desc
limit 10;


-- Query with range condition II.
\timing off
SELECT count(*) FROM gnis_column;
\timing on
\echo ';12a'
select id,version,lon,lat
from osm_poi_ch_1mio_column
where lon>47.22088 and lat>8.810778 and lon<47.22974 and lat<8.823223
order by version desc;
\echo ';12b'
select id,version,lon,lat
from osm_poi_ch_2mio_column
where lon>47.22088 and lat>8.810778 and lon<47.22974 and lat<8.823223
order by version desc;
\echo ';12c'
select id,version,lon,lat
from osm_poi_ch_3mio_column
where lon>47.22088 and lat>8.810778 and lon<47.22974 and lat<8.823223
order by version desc;
```

```
-- Query with group by
\timing off
SELECT count(*) FROM gnis_column;
\timing on
\echo ';13a'
select count(id), uid
from osm_poi_ch_1mio_column
group by uid having count(id)>1
order by 1 desc limit 10;
\echo ';13b'
select count(id), uid
from osm_poi_ch_2mio_column
group by uid having count(id)>1
order by 1 desc limit 10;
\echo ';13c'
select count(id), uid
from osm_poi_ch_3mio_column
group by uid having count(id)>1
order by 1 desc limit 10;

-- Query with 3 joins
-- Alle Restaurants mit id, name und Kô£¨¥nart (falls vorhanden):
\timing off
SELECT count(*) FROM gnis_column;
\timing on
\echo ';14a'
select e.id, av2.value as name, av3.value as cuisine, lon, lat
from osm_poi_ch_1mio_column as e
join osm_poi_tag_ch_column as av on e.id=av.id
left outer join osm_poi_tag_ch_column as av2 on e.id=av2.id
left outer join osm_poi_tag_ch_column as av3 on e.id=av3.id
where
  av.key='amenity' and av.value='restaurant'
  and av2.key='name'
  and av3.key='cuisine'
  and e.lon>47.22088 and e.lat>8.810778 and e.lon<47.22974 and
e.lat<8.823223
order by name
limit 10;
\echo ';14b'
select e.id, av2.value as name, av3.value as cuisine, lon, lat
from osm_poi_ch_2mio_column as e
join osm_poi_tag_ch_column as av on e.id=av.id
left outer join osm_poi_tag_ch_column as av2 on e.id=av2.id
left outer join osm_poi_tag_ch_column as av3 on e.id=av3.id
where
  av.key='amenity' and av.value='restaurant'
  and av2.key='name'
  and av3.key='cuisine'
  and e.lon>47.22088 and e.lat>8.810778 and e.lon<47.22974 and
e.lat<8.823223
order by name
limit 10;
```

```
\echo ';14c'
select e.id, av2.value as name, av3.value as cuisine, lon, lat
from osm_poi_ch_3mio_column as e
join osm_poi_tag_ch_column as av on e.id=av.id
left outer join osm_poi_tag_ch_column as av2 on e.id=av2.id
left outer join osm_poi_tag_ch_column as av3 on e.id=av3.id
where
  av.key='amenity' and av.value='restaurant'
  and av2.key='name'
  and av3.key='cuisine'
  and e.lon>47.22088 and e.lat>8.810778 and e.lon<47.22974 and
e.lat<8.823223
order by name
limit 10;

\echo '\nOk.'
```