Amazon Redshift
LAB: Table Layout and Schema Design

January, 2016

Table of Contents

Overview

This lab will walk you through some of the key aspects and considerations to have when designing a database for Amazon Redshift. The following is a high-level overview of this lab:

- Create Redshift Tables
- Test data compression
- Test distribution and sort keys
- Analyze storage efficiency and query performance

Before You Begin

If you don't already have an AWS account, you must sign up for one. You also need to download client tools and drivers in order to connect to the Amazon Redshift cluster. Please refer to Amazon Redshift Getting Started Guide (see http://docs.aws.amazon.com/redshift/latest/gsg/getting-started.html).

The examples in this lab are taken from the "TPC-H" database, a sample 100GB database, provided to any AWS Account on a public S3 bucket.

This lab expects you to have started a Redshift cluster in the US-EAST-1 (N. Virginia) region, and to be able to connect a PostgreSQL compatible client (e.g. SQL Workbench/J for example) to your cluster.

Prerequisites

1. Amazon Redshift does not provide or install any third-party tools or libraries. You will need to download the SQL Workbench/J from http://www.sql-workbench.net
2. To run SQL Workbench/J, you will also need to install the Java runtime environment
3. Once you have installed the SQL Workbench/J, you will need either a JDBC or ODBC driver. Amazon Redshift offers JDBC and ODBC drivers for download.
   NOTE: Previously, Amazon Redshift recommended PostgreSQL drivers for JDBC and ODBC; if you are currently using those drivers, we recommend moving to the new Amazon Redshift–specific drivers going forward. For more information about how to download the JDBC and ODBC drivers and configure connections to your cluster, see Configure a JDBC Connection and Configure an ODBC
4. Launch a Redshift cluster in the US EAST region from the AWS Redshift console (https://console.aws.amazon.com/redshift)
   a. Cluster Identifier – redshiftdemo
   b. Database Name – redshiftdemo
   c. Database Port – default 5439
   d. Master Username – awsuser
   e. Master Password – at least 8 characters, at least one uppercase and one lowercase letter with at least one number
   f. Cluster Configuration Node Type – dc1.8xlarge
   g. Cluster Type – Multi Node
   h. Number of Compute Nodes – 2
   i. Choose a VPC – Default VPC
   j. Cluster Subnet Group – default
   k. Publicly Accessible – Yes
   l. Choose a Public IP Address – No
   m. Availability Zone – No Preference
   n. VPC Security Groups – default
   o. Create CloudWatch Alarm – No
5. Setup the default security group to allow access to your EC2 client or laptop

Connecting to redshiftdemo database

1. Get the JDBC URL of your database from the Amazon Redshift console
2. To connect your client to the cluster, launch SQL Workbench/J and create a new profile with PostgresSQL driver and the JDBC URL of your database

Import Reference Data

In this first part, we will create two reference tables for managing user's geography, which will be used later in the lab.

Note: Before you proceed, ensure that your Redshift client is connected to the cluster.

1. Create tables: this script will create the tables with the default options in the database

   Copy the following statements to create the tables in the database.

```
CREATE TABLE region (
  r_regionkey int4 NOT NULL PRIMARY KEY                    ,
  r_name char(25) NOT NULL                                 ,
  r_comment varchar(152) NOT NULL
);

CREATE TABLE nation (
  n_nationkey int4 NOT NULL PRIMARY KEY                    ,
  n_name char(25) NOT NULL                                 ,
  n_regionkey int4 NOT NULL REFERENCES region(r_regionkey) ,
  n_comment varchar(152) NOT NULL
);
```

2. Import data into the tables: this script loads data from S3 into your database. Details on how the load process works will be given in a later lab

   Copy the following statements to load data into the tables. You must replace the <YOUR-ACCESS-KEY-ID>, and <YOUR-SECRET-ACCESS-KEY> with your own AWS account credentials.

```
COPY region FROM 's3://redshift-demo/tpc-h/100lzo/region/region.tbl.'
CREDENTIALS 'aws_access_key_id=<YOUR-ACCESS-KEY-
ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>' LZOP;

COPY nation FROM 's3://redshift-demo/tpc-h/100lzo/nation/nation.tbl.'
CREDENTIALS 'aws_access_key_id=<YOUR-ACCESS-KEY-
ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>' LZOP;
```


Compressing and denormalizing: Customer

Standard layout

Redshift operates on high amounts of data. In order to optimize Redshift workloads, one of the key principles is to lower the amount of data stored. Diminishing this volume is achieved by using a set of compression algorithms on the data stored. Instead of working on entire rows of data, containing values of different types and function, Redshift operates in a columnar fashion, this gives the opportunity to implement algorithms that can operate on single columns of data, thus greatly enhancing their efficiency. In this example we will load data into a table from an external file and test what compression scheme can be used.

You can apply compression encodings to columns in tables manually, based on your own evaluation of the data, or you can use the COPY command to analyze and apply compression automatically. We strongly recommend using the COPY command to apply automatic compression. However in this section of this lab we will turn off the automatic compression, do it manually first to show you the process of analyzing tables and the performance gains of using correct compression.

NOTE: You can keep track of each query in the schema_design_lab_worksheet.xlsx spreadsheet.

1. Create the customer table using the default settings, only specifying DISTKEY and SORTKEY to the customer key.

   Copy the following statements to create the table in the database.

```
CREATE TABLE customer_v1 (
  c_custkey int8 NOT NULL DISTKEY SORTKEY PRIMARY KEY ENCODE RAW     ,
  c_name varchar(25) NOT NULL ENCODE RAW                             ,
  c_address varchar(40) NOT NULL ENCODE RAW                          ,
  c_nationkey int4 NOT NULL REFERENCES nation(n_nationkey) ENCODE RAW,
  c_phone char(15) NOT NULL ENCODE RAW                               ,
  c_acctbal numeric(12,2) NOT NULL ENCODE RAW                        ,
  c_mktsegment char(10) NOT NULL ENCODE RAW                          ,
  c_comment varchar(117) NOT NULL ENCODE RAW
);
```

2. Import data from S3 into this table.

   Copy the following statements to load data into the table. You must replace the <YOUR-ACCESS-KEY-ID>, and <YOUR-SECRET-ACCESS-KEY> with your own AWS account credentials.

```
COPY customer_v1 FROM 's3://redshift-demo/tpc-
h/100lzo/customer/customer.tbl.' COMPUPDATE OFF CREDENTIALS
'aws_access_key_id=<YOUR-ACCESS-KEY-ID>;aws_secret_access_key=<YOUR-SECRET-
ACCESS-KEY>' LZOP;
```


3. "VACUUM" the table. I.e. ensure data is correctly organized on disk to maximize performances. This command will described in more details in the "Data Loading part of the workshop".

   Copy the following statements to enhance storage of the table.

```
VACUUM customer_v1;
```

4. Analyze the table layout. This query analyses the data and builds statistics for a table, for optimizing query execution times.

   Copy the following statements to analyze the in the database.

```
ANALYZE customer_v1;
```

5. Analyze the storage optimization options for this table. You can yourself choose the compression scheme you want or leave Redshift determine one for you. This query analyzes the data in the table and presents recommendations on how to improve storage and performances for this table. The result of this statement is important and gives you insights on how to optimize storage based on the current real data stored in the table.

   Copy the following statements to analyze compression of the table.

```
ANALYZE COMPRESSION customer_v1;
```

Compression optimization

Based on the results from the previous "ANALYZE COMPRESSION" command, we can insert the same data in a new table and analyze the difference in storage.

1. Create the new customer table using the hints provided by the last compression analysis.

   Copy the following statements to create the table in the database.

```
CREATE TABLE customer_v2 (
  c_custkey int8 NOT NULL ENCODE DELTA DISTKEY SORTKEY PRIMARY KEY        ,
  c_name varchar(25) NOT NULL ENCODE LZO                                  ,
  c_address varchar(40) NOT NULL ENCODE LZO                               ,
  c_nationkey int4 NOT NULL ENCODE DELTA REFERENCES nation(n_nationkey),
  c_phone char(15) NOT NULL ENCODE LZO                                    ,
  c_acctbal numeric(12,2) NOT NULL ENCODE LZO                             ,
  c_mktsegment char(10) NOT NULL ENCODE BYTEDICT                          ,
  c_comment varchar(117) NOT NULL ENCODE TEXT255
);
```


2. Import data from the previous table into this table.

   Copy the following statements to load data into the table.

```
INSERT INTO customer_v2(c_custkey, c_name, c_address, c_nationkey, c_phone,
c_acctbal, c_mktsegment, c_comment)
SELECT c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal,
c_mktsegment, c_comment
FROM customer_v1;
```

3. Clean-up storage and build statistics for the table.

   Copy the following statements to enhance storage and analyze statistics of the table.

```
VACUUM customer_v2;
ANALYZE customer_v2;
```

4. Analyze the storage space for these tables, before and after compression. The table stores by column the amount of storage used in MiB. You should see about a 50% savings on the storage of the second table compared to first. This query gives you the storage requirements per column for each table, then the total storage for the table (repeated identically on each line).

   Copy the following statements to compare storage space of both tables.

```
SELECT
  CAST(d.attname AS CHAR(50)),
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v1,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v2,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v1,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v2
FROM
(
  SELECT relname, attname, attnum - 1 as colid
  FROM pg_class t
  INNER JOIN pg_attribute a ON a.attrelid = t.oid
  WHERE t.relname LIKE 'customer\_v%'
) d
INNER JOIN
(
  SELECT name, col, MAX(blocknum) AS size_in_mb
  FROM stv_blocklist b
  INNER JOIN stv_tbl_perm p ON b.tbl=p.id
  GROUP BY name, col
) b
ON d.relname = b.name AND d.colid = b.col
GROUP BY d.attname
ORDER BY d.attname;
```


Data denormalization

Compression allows the storage of "reference" data inside the fact table, removing some of the needs for "star" or "snowflakes" database designs for storage optimization. This part will do that with the nation and region information for customer, integrating these columns in the source table and analyzing the differences.

1. Create the new customer table, denormalizing nation and region names to be included directly in the customer table.

   Copy the following statements to create the table in the database.

```
CREATE TABLE customer_v3 (
  c_custkey int8 NOT NULL ENCODE DELTA DISTKEY SORTKEY PRIMARY KEY,
  c_name varchar(25) NOT NULL ENCODE LZO                        ,
  c_address varchar(40) NOT NULL ENCODE LZO                       ,
  c_nationname char(25) NOT NULL ENCODE BYTEDICT                   ,
  c_regionname char(25) NOT NULL ENCODE BYTEDICT                   ,
  c_phone char(15) NOT NULL ENCODE LZO                              ,
  c_acctbal numeric(12,2) NOT NULL ENCODE LZO                     ,
  c_mktsegment char(10) NOT NULL ENCODE BYTEDICT                    ,
  c_comment varchar(117) NOT NULL ENCODE TEXT255
);
```

2. Import data from the previous table into this table. Note the joins to flatten the schema.

   Copy the following statements to load data into the table.

```
INSERT INTO customer_v3(c_custkey, c_name, c_address, c_nationname,
c_regionname, c_phone, c_acctbal, c_mktsegment, c_comment)
SELECT c_custkey, c_name, c_address, n_name, r_name, c_phone, c_acctbal,
c_mktsegment, c_comment
FROM customer_v2
INNER JOIN nation ON c_nationkey = n_nationkey
INNER JOIN region ON n_regionkey = r_regionkey;
```

3. Clean-up storage and build statistics for the table.

   Copy the following statements to enhance storage and analyze statistics of the table.

```
VACUUM customer_v3;
ANALYZE customer_v3;
```


4. Analyze the difference in storage space for these three versions of the customer table. Adding the columns just added the space required for this compressed column, and with the compression algorithm used, the difference is less than 2% the size of the table. This query gives you the storage requirements per column for each table, then the total storage for the table (repeated identically on each line).

   Copy the following statements to compare storage space of all three tables.

```
SELECT
  CAST(d.attname AS CHAR(50)),
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v1,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v2,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v3%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v3,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v1,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v2,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v3%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v3
FROM
(
  SELECT relname, attname, attnum - 1 as colid
  FROM pg_class t
  INNER JOIN pg_attribute a ON a.attrelid = t.oid
  WHERE t.relname LIKE 'customer\_v%'
) d
INNER JOIN
(
  SELECT name, col, MAX(blocknum) AS size_in_mb
  FROM stv_blocklist b
  INNER JOIN stv_tbl_perm p ON b.tbl=p.id
  GROUP BY name, col
) b
ON d.relname = b.name AND d.colid = b.col
GROUP BY d.attname
ORDER BY d.attname;
```

Queries

While we won't cover the details of Redshift queries, this section gives an example of a single query processed on all three tables. Data warehousing systems being designed for WORM (Write Once Read Many) type of workloads, the optimization of the table must be made knowing what the queries that will run on it will be. Redshift proposes system tables to analyze query performances, we will start using them.

1. Get customers from the "Asia" region from the first table.

   Copy the following statements to query the customers.


```
SELECT COUNT(c_custkey)
FROM customer_v1 c
INNER JOIN nation n ON c.c_nationkey = n.n_nationkey
INNER JOIN region r ON n.n_regionkey = r.r_regionkey
WHERE r.r_name = 'ASIA';
```

2. From the second table.

   Copy the following statements to query the customers.

```
SELECT COUNT(c_custkey)
FROM customer_v2 c
INNER JOIN nation n ON c.c_nationkey = n.n_nationkey
INNER JOIN region r ON n.n_regionkey = r.r_regionkey
WHERE r.r_name = 'ASIA';
```

3. And from the third table.

   Copy the following statements to query the customers.

```
SELECT COUNT(c_custkey)
FROM customer_v3 c
WHERE c.c_regionname = 'ASIA';
```

4. Analyze the performance of each query. This query gets the 3 last queries ran against the database, and you should get your three last queries. The first executions may be comparably equal on the three queries, but if you repeat the execution multiple times (which is the case in a data warehousing environment), you will see that the query targeting the compressed table (second table) performs up to about 40% better than the one targeting the uncompressed table (first table), and that the query targeting the denormalized table (third table) has speed gains up to 50% compared to the one on the second table (or approximately a 75% better than the one on the uncompressed table). (*The numbers may vary depending on the cluster topology*)

   Copy the following statements to analyze query performance.

```
SELECT query, TRIM(querytxt) as SQL, starttime, endtime, DATEDIFF(microsecs,
starttime, endtime) AS duration
FROM STL_QUERY
WHERE "database" <> 'dev'
ORDER BY query DESC
LIMIT 3;
```


Distributing and sorting: Orders

Standard layout

This example will import order data from an external source. We will iterate on different table designs to check the impact in terms of storage and query execution speed.

1. Create the orders table using the default settings, only specifying DISTKEY and SORTKEY to the order key.

   Copy the following statements to create the table in the database.

```
CREATE TABLE orders_v1 (
  o_orderkey int8 NOT NULL DISTKEY SORTKEY PRIMARY KEY     ,
  o_custkey int8 NOT NULL REFERENCES customer_v3(c_custkey),
  o_orderstatus char(1) NOT NULL                           ,
  o_totalprice numeric(12,2) NOT NULL                      ,
  o_orderdate date NOT NULL                                ,
  o_orderpriority char(15) NOT NULL                        ,
  o_clerk char(15) NOT NULL                                ,
  o_shippriority int4 NOT NULL                             ,
  o_comment varchar(79) NOT NULL
);
```

2. Import data from S3 into this table.

   Copy the following statements to load data into the table. You must replace the <YOUR-ACCESS-KEY-ID>, and <YOUR-SECRET-ACCESS-KEY> with your own AWS account credentials. Note: May take about 5 mins with 2 dc1.large nodes

```
COPY orders_v1 FROM 's3://redshift-demo/tpc-h/100lzo/orders/orders.tbl.'
COMPUPDATE OFF CREDENTIALS 'aws_access_key_id=<YOUR-ACCESS-KEY-
ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>' LZOP;
```

3. Vacuum and analyze storage, and check for compression options.

   Copy the following statements to analyze and optimize the table.

```
VACUUM orders_v1;
ANALYZE orders_v1;
ANALYZE COMPRESSION orders_v1;
```


Compression

The same table will be loaded with compression activated this time through automated compression via COPY by removing COMPUPDATE OFF (by default COPY will compress unless COMPUTDATE OFF is used) from the COPY command.

1. Create the orders table using the previous compression propositions, with DISTKEY and SORTKEY to the order key.

   Copy the following statements to create the table in the database.

```
CREATE TABLE orders_v2 (
  o_orderkey int8 NOT NULL DISTKEY SORTKEY PRIMARY KEY     ,
  o_custkey int8 NOT NULL REFERENCES customer_v3(c_custkey),
  o_orderstatus char(1) NOT NULL                           ,
  o_totalprice numeric(12,2) NOT NULL                      ,
  o_orderdate date NOT NULL                                ,
  o_orderpriority char(15) NOT NULL                        ,
  o_clerk char(15) NOT NULL                                ,
  o_shippriority int4 NOT NULL                             ,
  o_comment varchar(79) NOT NULL
);
```

2. Import data using the COPY command with automatic compression.

   Copy the following statements to load data into the table.

   Note: May take about 7 mins with 2 dc1.large nodes

```
COPY orders_v2 FROM 's3://redshift-demo/tpc-h/100lzo/orders/orders.tbl.'
CREDENTIALS 'aws_access_key_id=<YOUR-ACCESS-KEY-
ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>' LZOP;
```

3. Vacuum and analyze storage.

   Copy the following statements to analyze and optimize the table.

```
VACUUM orders_v2;
ANALYZE orders_v2;
```

Distributing data

One of the key features enabling Redshift's scale is the possibility to slice the data dynamically across nodes. This can be done evenly or in a round-robin fashion (default), by a distribution key (or column) or an ALL distribution which puts all data in all slices. These options give you the ability to spread out data in a distributed cluster to maximize the parallelization of the queries.

For fast query performance, we recommended using a distribution key that will be used in regularly joined tables, allowing Redshift to co-locate the data of these different entities, reducing IO and network exchanges. We will explore different distribution methods and observe impact to the data and query performance.

Redshift also uses a specific Sort column to know in advance what values of a column are in a given block, and to skip reading that entire block if the values it contains don't fall into the range of a query.

In this sample, queries are based on customer related information (region), making the customer key a good fit for distribution key, and the filters are made on order date ranges, so using it as a sort key helps execution.

1. Create the orders table with default settings, this time changing DISTKEY to customer key and SORTKEY to order date.

   Copy the following statements to create the table in the database.

```
CREATE TABLE orders_v3 (
  o_orderkey int8 NOT NULL PRIMARY KEY                                ,
  o_custkey int8 NOT NULL DISTKEY REFERENCES customer_v3(c_custkey),
  o_orderstatus char(1) NOT NULL                                      ,
  o_totalprice numeric(12,2) NOT NULL                                 ,
  o_orderdate date NOT NULL SORTKEY                                   ,
  o_orderpriority char(15) NOT NULL                                   ,
  o_clerk char(15) NOT NULL                                           ,
  o_shippriority int4 NOT NULL                                        ,
  o_comment varchar(79) NOT NULL
);
```

2. Import data from the existing table into this table.

   Copy the following statements to load data into the table.
   Note: May take about 3 mins with 2 dc1.large nodes

```
INSERT INTO orders_v3(o_orderkey, o_custkey, o_orderstatus, o_totalprice,
o_orderdate,o_orderpriority, o_clerk, o_shippriority, o_comment)
SELECT o_orderkey, o_custkey, o_orderstatus, o_totalprice,
o_orderdate,o_orderpriority, o_clerk, o_shippriority, o_comment
FROM orders_v1;
```

3. Vacuum and analyze storage.

   Copy the following statements to analyze and optimize the table.

```
VACUUM orders_v3;
ANALYZE orders_v3;
```


4. Analyze compression options. You will see that they have changed from the previous entries. Compression depends directly on the data as it is stored on disk, and storage is modified by distribution and sort options.

   Copy the following statements to analyze compression for the table.

```
ANALYZE COMPRESSION orders_v3;
```

Altogether

This last step will use the new distribution and sort keys, and the compression settings proposed by Redshift.

1. Create the orders table using the recommended compression propositions, keeping DISTKEY to customer key and SORTKEY to order date.

   Copy the following statements to create the table in the database.

```
CREATE TABLE orders_v4 (
  o_orderkey int8 NOT NULL PRIMARY KEY                                ,
  o_custkey int8 NOT NULL DISTKEY REFERENCES customer_v3(c_custkey),
  o_orderstatus char(1) NOT NULL                                      ,
  o_totalprice numeric(12,2) NOT NULL                                 ,
  o_orderdate date NOT NULL SORTKEY                                   ,
  o_orderpriority char(15) NOT NULL                                   ,
  o_clerk char(15) NOT NULL                                           ,
  o_shippriority int4 NOT NULL                                        ,
  o_comment varchar(79) NOT NULL
);
```

2. Import data using COPY command with automatic compression.

   Copy the following statements to load data into the table.
   Note: May take about 6 mins with 2 dc1.large nodes

```
COPY orders_v4 FROM 's3://redshift-demo/tpc-h/100lzo/orders/orders.tbl.'
CREDENTIALS 'aws_access_key_id=<YOUR-ACCESS-KEY-
ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>' LZOP;
```

3. Vacuum and analyze storage.

   Copy the following statements to analyze and optimize the table.


```
VACUUM orders_v4;
ANALYZE orders_v4;
```

4. Finally, lets do one more version using ALL distribution type to put a copy of all the data in every slice of the cluster creating the largest data foot print but putting this data as close to all other data as possible.

```
CREATE TABLE orders_v5 (
  o_orderkey int8 NOT NULL                                          ,
  o_custkey int8 NOT NULL REFERENCES customer_v3(c_custkey)         ,
  o_orderstatus char(1) NOT NULL                                    ,
  o_totalprice numeric(12,2) NOT NULL                               ,
  o_orderdate date NOT NULL SORTKEY                                 ,
  o_orderpriority char(15) NOT NULL                                 ,
  o_clerk char(15) NOT NULL                                         ,
  o_shippriority int4 NOT NULL                                      ,
  o_comment varchar(79) NOT NULL                                    ,
  primary key (o_orderkey)) diststyle all;
```

5. Import data using COPY command with automatic compression.

Copy the following statements to load data into the table.
Note: May take about 9 mins with 2 dc1.large nodes

```
COPY orders_v5 FROM 's3://redshift-demo/tpc-
h/100lzo/orders/orders.tbl.' CREDENTIALS 'aws_access_key_id=<YOUR-
ACCESS-KEY-ID>;aws_secret_access_key=<YOUR-SECRET-ACCESS-KEY>'
LZOP;
```

6. Vacuum and analyze storage.

Copy the following statements to analyze and optimize the table.

```
VACUUM orders_v5;
ANALYZE orders_v5;
```

Storage analysis

As for the customers, this query will analyze the storage used by the five representations of the orders table.

1. Analyze the difference in storage space for these five versions of the order table. Compression allows a 50% to 60% storage reduction on the data. The fourth version is slightly bigger (2% to 3%) than the second version, because compression schemes have changed. The fifth version is the largest amount of data as it stores all data in this table in all slices of the cluster. If you wish

to learn more detailed information about distributing data in a redshift cluster please see http://docs.aws.amazon.com/redshift/latest/dg/t_Distributing_data.html.

This query gives you the storage requirements per column for each table, then the total storage for the table (repeated identically on each line).

Copy the following statements to compare storage space of all four tables.

```
SELECT
  CAST(d.attname AS CHAR(50)),
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v1,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v2,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v3%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v3,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v4%' THEN b.size_in_mb
ELSE 0 END) AS size_in_mb_v4,
  SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v5%' THEN b.size_in_mb
ELSE  0 END) AS size_in_mb_v5,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v1%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v1,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v2%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v2,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v3%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v3,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v4%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v4,
  SUM(SUM(CASE WHEN CAST(d.relname AS CHAR(50)) LIKE '%v5%' THEN b.size_in_mb
ELSE 0 END)) OVER () AS total_mb_v5
FROM
(
  SELECT relname, attname, attnum - 1 as colid
  FROM pg_class t
  INNER JOIN pg_attribute a ON a.attrelid = t.oid
  WHERE t.relname LIKE 'orders\_v%'
) d
INNER JOIN
(
  SELECT name, col, MAX(blocknum) AS size_in_mb
  FROM stv_blocklist b
  INNER JOIN stv_tbl_perm p ON b.tbl=p.id
  GROUP BY name, col
) b
ON d.relname = b.name AND d.colid = b.col
GROUP BY d.attname
ORDER BY d.attname;
```

Queries

The query execution speed is also impacted by the compression and distribution settings. This last part will issue the same query on the five versions of the order table, and analyze the time taken to execute these queries.

1. Get, for the year 1995, some information on the orders passed by the customers depending on their market segment, in Asia. This query is for the first table.

   Copy the following statements to query the first table.

```
SELECT c_mktsegment, COUNT(o_orderkey) AS orders_count, AVG(o_totalprice) AS
medium_amount, SUM(o_totalprice) AS orders_revenue
FROM orders_v1 o
INNER JOIN customer_v3 c ON o.o_custkey = c.c_custkey
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-12-31' AND c_regionname =
'ASIA'
GROUP BY c_mktsegment;
```

2. Same query for the second table.

   Copy the following statements to query the table.

```
SELECT c_mktsegment, COUNT(o_orderkey) AS orders_count, AVG(o_totalprice) AS
medium_amount, SUM(o_totalprice) AS orders_revenue
FROM orders_v2 o
INNER JOIN customer_v3 c ON o.o_custkey = c.c_custkey
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-12-31' AND c_regionname =
'ASIA'
GROUP BY c_mktsegment;
```

3. Again for the third table. You will notice that the order of results has changed. This is due to the change in sorting and distribution, since we did not order the resultset (no ORDER clause), the "natural" (storage) order applies.

   Copy the following statements to query the table.

```
SELECT c_mktsegment, COUNT(o_orderkey) AS orders_count, AVG(o_totalprice) AS
medium_amount, SUM(o_totalprice) AS orders_revenue
FROM orders_v3 o
INNER JOIN customer_v3 c ON o.o_custkey = c.c_custkey
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-12-31' AND c_regionname =
'ASIA'
GROUP BY c_mktsegment;
```

4. Again for the fourth.

   Copy the following statements to query the table.

```
SELECT c_mktsegment, COUNT(o_orderkey) AS orders_count, AVG(o_totalprice) AS
medium_amount, SUM(o_totalprice) AS orders_revenue
FROM orders_v4 o
```



```
INNER JOIN customer_v3 c ON o.o_custkey = c.c_custkey
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-12-31' AND c_regionname =
'ASIA'
GROUP BY c_mktsegment;
```

5. Finally for the fifth and final table.

   Copy the following statements to query the table.

```
SELECT c_mktsegment, COUNT(o_orderkey) AS orders_count, AVG(o_totalprice)
AS medium_amount, SUM(o_totalprice) AS orders_revenue
FROM orders_v5 o
INNER JOIN customer_v3 c ON o.o_custkey = c.c_custkey
WHERE o_orderdate BETWEEN '1995-01-01' AND '1995-12-31' AND c_regionname =
'ASIA'
GROUP BY c_mktsegment;
```

6. Analyze the performances of each query. This query gets the 5 last queries ran against the database, and you should get your five last queries. The results go up to around 75% query time improvement with the right distribution, sort and compression schemes (orders_v4 vs orders_v5). It is important to note that the uncompressed version with the adapted distribution and sorting (orders_v3) performs constantly better than the compressed version with primary key as distribution key (orders_v2). The ALL distribution (v5) should really be used only if a dimension table cannot be collocated with the fact table or other important joining tables, you can improve query performance significantly by distributing the entire table to all of the nodes. Using ALL distribution multiplies storage space requirements and increases load times and maintenance operations, so you should weigh all factors before choosing ALL distribution. (*The numbers will vary depending on the cluster topology*)

   Copy the following statements to analyze query performance results.

```
SELECT query, TRIM(querytxt) as SQL, starttime, endtime, DATEDIFF(microsecs,
starttime, endtime) AS duration
FROM STL_QUERY
WHERE "database" <> 'dev'
ORDER BY query DESC
LIMIT 5;
```

Before You Leave

If you are done using your cluster, please think about decommissioning it to avoid having to pay for unused resources.

