Masaryk University
Faculty of Informatics

Analysis of Vertica Database Designer

Master's thesis

Bc. Martin Zbořil

Brno, spring 2017

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Zbořil

Advisor: doc. RNDr. Vlastislav Dohnal, Ph.D.

Acknowledgement

I would like to thank my advisor doc. RNDr. Vlastislav Dohnal, Ph.D. for many useful comments, valuable insights, recommendations, and his patience. I would also like to thank my family and Adéla Šrámková for supporting me throughout writing this thesis.

I would like to express my gratitude to GoodData s.r.o. that enabled me to be a part of the team and work on the thesis during my internship. I would like to thank namely to Jan Soubusta, Mgr. Josef Pojsl and Ing. Vladimír Vacula.

Abstract

The major goal of this thesis is to analyze the Database Designer tool that optimizes physical schema in the Vertica database system. Database Designer might be beneficial for Vertica users, but optimality of the tool must be uncovered before its usage in any critical database environment, like production servers of companies. In the theoretical part, the aim of the thesis is to describe fundamentals of column-oriented database systems and Vertica specifics. Furthermore, principles of relational database tuning with the focus on Vertica is included in the thesis. The Database Designer analysis was done in two ways. The first one included finding whether the tool influences database availability. The second one included designing and implementing a tool that targets suboptimal designs generated by Database Designer. Based on the data measured by the implemented tool, bottlenecks of Database Designer were found. This thesis is done in cooperation with GoodData, s.r.o.

Keywords

Column-oriented database system, Vertica, projection, Database Designer, database availability, database tuning, testing tool

Contents

1 **Introduction** . . . 3
2 **Column-oriented database system** . . . 5
3 **Vertica** . . . 10
- 3.1 *Glossary* . . . 10
- 3.2 *Characteristics* . . . 12
  - 3.2.1 Projections . . . 13
  - 3.2.2 Encoding . . . 16
  - 3.2.3 Segmentation . . . 16
  - 3.2.4 Partitioning . . . 17
  - 3.2.5 High Availability . . . 17
- 3.3 *Locks* . . . 19
- 3.4 *System tables* . . . 19

4 **Database tuning** . . . 21
- 4.1 *General database tuning* . . . 21
- 4.2 *Database tuning in Vertica* . . . 23
  - 4.2.1 Query monitoring . . . 25
- 4.3 *Database Designer* . . . 28
  - 4.3.1 Functions of tool . . . 28
  - 4.3.2 Designing step . . . 30

5 **DBPerfComp tool** . . . 32
- 5.1 *Test goal* . . . 32
- 5.2 *Tool design* . . . 33
- 5.3 *Output* . . . 41
- 5.4 *Documentation* . . . 42
- 5.5 *Directory layout* . . . 48

6 **Analysis** . . . 49
- 6.1 *TPC-H model* . . . 49
- 6.2 *Hardware and software* . . . 50
- 6.3 *Database Designer's influence on database objects* . . . 51
  - 6.3.1 Tables location . . . 52
  - 6.3.2 TPC-H queries, Database Designer and different catalog size . . . 54
  - 6.3.3 Steps of designing process . . . 54
  - 6.3.4 Different catalog size . . . 57

7 **Data evaluation** . . . 63
- 7.1 *Test setting* . . . 63
- 7.2 *Results* . . . 65

7.2.1 Cluster 1 65
7.2.2 Cluster 2 73
7.3 *Query 5* 74
7.4 *Projection definition* 76
7.5 *Query plan* 78
7.6 *Database operations* 80
7.7 *Manual optimization* 83
7.8 *Results summary* 84
8 **Conclusion** 86
A **Lock modes** 91
B **Parameters of main function of Database Designer** 92
C **Output tables of DBPerfComp** 93
C.1 *Table – monitoring of query runs* 93
C.2 *Main table – processed and measured data* 94
C.3 *Table – queries order in query workloads* 95
C.4 *Logs* 96
D **Query for investigation of Database Designer influence on database objects** 97
E **Locks – CRUD operations** 98
F **TPC-H queries created for the experiments** 99
F.1 *Query 23* 99
F.2 *Query 24* 100
G **List of query workloads with timeout** 101
H **Query workloads – ordered by ratio** 102
I **Cluster2 – results** 104
J **Query for investigation of a query plan** 108
K **Simplified Query 5** 110
L **Query for investigation of query operations** 111
M **Bug report** 112

1 Introduction

Processing of large volumes of data plays an important role in data warehouses and other systems for data analytics. Due to fast-growing volumes of data, column-oriented database systems have been gaining more importance in data analytics then before. S. G. Yaman wrote about fundamentals of column-oriented database systems in his paper *Introduction to Column-Oriented Database Systems*[1] as follows:

> *"In a column-store database, each column is stored contiguously on a separate location on a disk. The values stored in the columns are densely packed and compressed to improve read efficiency. Column-store databases perform faster than traditional database systems, since they are more I/O efficient for read-only queries. In that manner, column-scanners are different from row-scanners, since column-scanners translate value position information into disk locations and they combine and reconstruct the tuples from different columns.*
>
> *Column-store systems include column-oriented physical design, and it is observed that due to superior CPU and cache performance (in addition to reduced I/O), they can perform better compared to commercial and open source row-store databases on benchmarks. Additionally, they include optimizations for direct operation on compressed data."*

The roots of these column-oriented database systems reach the 1970s, but its expansion is dated to the 2000s when MonetDB and C-Store database systems were introduced to the public.[1] According to the *DB-Engines* portal[2], only one column-oriented database system was positioned in the top 10 most popular database engines in April 2017, and it was MonetDB. Next examples of column-oriented database systems are Vertica, Sybase IQ, Greenplum and InfiniDB as commercial column-oriented database systems; C-Store and LucidDB as open-source column-oriented database systems.[3]

The first goal of this thesis is to study column-oriented database system with the special focus on Vertica. The second goal is to analyze performance and behavior of Vertica's Database Designer.

The thesis is organized as follows. Chapter 2 is dedicated to principles of column-oriented database systems and Chapter 3 to specifics of Vertica database system. The tuning principles of relational databases and Vertica are described in Chapter 4, but only as a background, not being the focus of this thesis. Chapter 5 introduces the implemented DBPerfComp tool as the main contribution of this thesis; the goal, design, and documentation of that tool are included in this chapter. Chapter 6 presents measurements performed in order to analyze the influence of Database Designer on other objects in a database, i.e. whether Database Designer affects database availability. Data measured by the implemented tool is evaluated in Chapter 7; this evaluation involves detection of Database Designer bottlenecks.

2 Column-oriented database system

This chapter introduces a database concept that differs from the traditional one that stores data by rows. The concept, which is described in subsequent pages, is called *column-oriented database system*.

This thesis focuses on the relational database system. The relational model tranforms a business into a set of tables, their attributes, types of attributes and constraints. An example of the relational model is shown in Figure 2.1. In principle, there are two approaches of storing relational data in tables – row-oriented and column-oriented storage. Each of them has its advantages and disadvantages, which is the subject of the following sections.

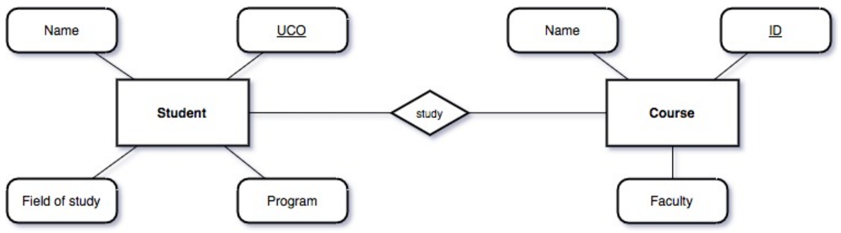


Figure 2.1: An example of relational model.

Storing tables of a relational model

The difference between storing tables by columns and by rows is shown in Figure 2.2 where the *tickstore* table includes a sample of attributes and entities. As it is shown in the upper part of the figure, column-oriented database systems store tables by columns where one column represents one table attribute. This approach is illustrated with vertical bars. The bottom part of the figure shows row-oriented database systems. This traditional approach stores tables by rows where one row represents all information stored about one entity. In general, records are stored one-by-one consecutively in one file, which is exemplified by horizontal bars.

To present merits of individual organizations, query evaluation efficiency must be tackled. Figure 2.2 includes *SELECT* query that takes average price for the specific symbol and date. Column-oriented database system works only with attributes that are specified the query, i.e. *SYMBOL*, *DATE* and *PRICE* attributes that have the bars with green lines in the figure. The bars with red lines are skipped since the attributes are not used in the query. This approach is possible because each attribute is stored in a separate file. At the row-oriented database system, the query must

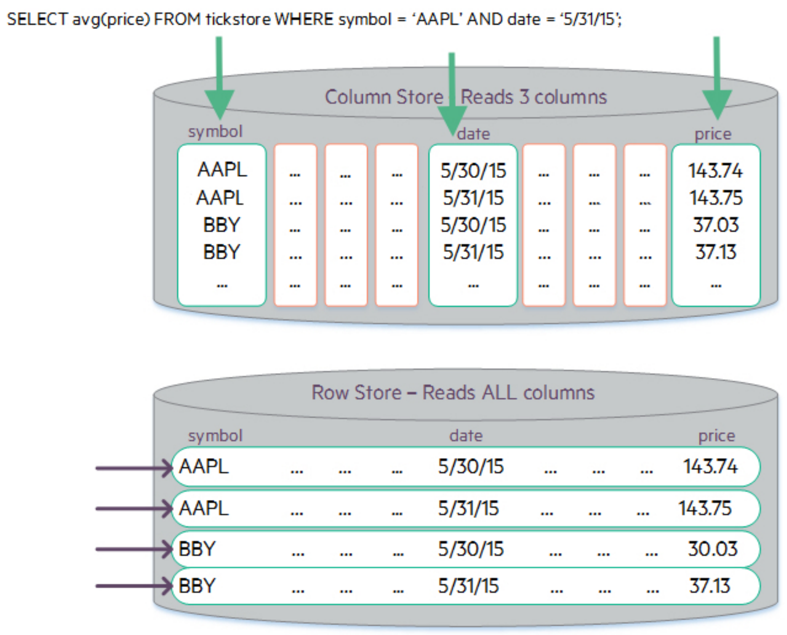

SELECT avg(price) FROM tickstore WHERE symbol = 'AAPL' AND date = '5/31/15';
Column Store – Reads 3 columns
symbol
date
price
AAPL AAPL BBY BBY ...
5/30/15 5/31/15 5/30/15 5/31/15 ...
143.74 143.75 37.03 37.13 ...
Row Store – Reads ALL columns
AAPL ... ... 5/30/15 ... ... 143.74
AAPL ... ... 5/31/15 ... ... 143.75
BBY ... ... 5/30/15 ... ... 30.03
BBY ... ... 5/31/15 ... ... 37.13

Figure 2.2: Comparison of column-oriented and row-oriented database systems.[4]

go through all attributes, since all attributes of each record are stored in one file. This situation is pictured with the green lines of the horizontal bars. The majority of attributes may not be interesting for the query; nevertheless, the query must go through them as well.[5]

Advantages and disadvantages

Column-oriented database systems are very efficient in read-only queries (preferably selecting few attributes), aggregation operations and updating a particular column of entities. The reason was already described; queries use only columns that are needed, so no redundant data is processed. Moreover, aggregation functions (e.g. AVG, SUM, COUNT, MAX) need incomparably fewer resources, since they operate only on particular columns. For that reason, column-oriented database systems are used for analytic purposes and are frequent in data warehouses and Online Analytical Processing (OLAP). A column-oriented approach is efficient mainly when a vast number of rows and a small number of columns are processed. Examples of queries efficient in column-oriented database systems are:

- *SELECT name FROM customer WHERE country = 'France'*
- *SELECT test_number, COUNT(response_time), AVG(response_time), SUM(response_time), MIN(response_time), MAX(response_time) FROM measurements WHERE date = '04-05-2017' GROUP BY test_number*
- *SELECT order_status, COUNT(*), SUM(totalprice) FROM orders WHERE order_priority = 1 GROUP BY order_status*

Row-oriented database systems are efficient in a case of retrieving data from more than a few columns, because they read complete rows. They are also more efficient in inserting, updating of many columns and deleting entities, since they work only with particular rows. For example, inserting of one entity generally requires only one disk access, since the entity is stored only in one file. In contrast, insertion of one entity into column-oriented database systems requires the same amount of disk access as the number of columns in the used table. In general, row-oriented database systems are more efficient when a small number of rows and a vast number of columns are processed. Row-oriented database systems are frequently used in Online Transaction Processing (OLTP) systems and are not optimal for analytical (OLAP) systems.[6] Examples of queries efficient in row-oriented database systems are:

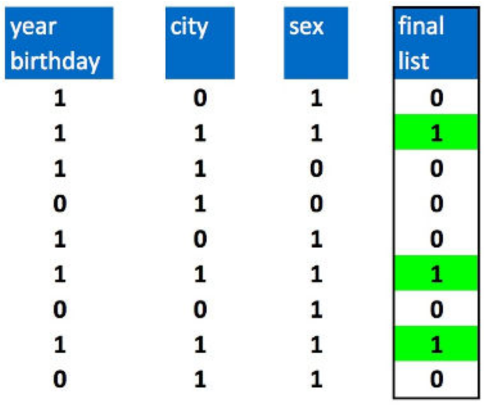
- *INSERT INTO region (regionkey, name, comment) VALUES ('1', 'EUROPE', 'many countries')*
- *UPDATE course SET name = 'Column-oriented database systems', credits = 3, completion = 'exam' WHERE id = 21*
- *DELETE FROM student WHERE uco = '13579'*

Performance optimization

Late materialization and compression are the most important approaches for performance optimization in column-oriented database systems. In contrast to row-oriented database systems where all attributes of one row are stored together, column-oriented database systems store particular attributes of entities in many different places. Thus, many files with columns are opened during execution of any query in column-oriented database systems. For that reason, a vast number of data in the form of retrieved tuples may be stored in memory, and database operations are performed on this data then; this is a case of naive column-oriented database systems. Late materialization is used in recent column-oriented database systems to improve a performance of data retrieving. A principle of late materialization is that data is kept in columns as long as possible, and queries operate directly on the columns then. For this purpose, *position lists* are often used for particular operations of given query. Position lists may have many forms; in the next example, a position list has a form of a sequence of bits (1/0). A query that may be used for showing exemplary usage of these lists looks like:

```
SELECT name, phone_number
FROM customer
WHERE year_birthday = 1992 AND city = ’Brno’ AND sex = ’Male’
```

For each predicate, one position list is created on an appropriate column. If the *n*-th record of the used column passes the predicate, the *n*-th item of the position list is set to *1*; otherwise, the value of the *n*-th item in the position list is set to *0*. Then, all position lists are intersected, and one final position list is created. In the end, this final position list is applied on the columns used for selection, and appropriate data is retrieved. The example of position lists for mentioned query is in Figure 2.3.



year birthday	city	sex	final list
1	0	1	0
1	1	1	1
1	1	0	0
0	1	0	0
1	0	1	0
1	1	1	1
0	0	1	0
1	1	1	1
0	1	1	0

Figure 2.3: An example of position lists.

Compression of data in particular columns is one of the greatest advantages of column-oriented database systems, since compression has a significant impact on performance. Often, all records of one particular attribute have a similar form (e.g. Male/Female) or other relationship (e.g. a sequence). In that case, this attribute is a perfect candidate for compression in column-oriented database systems. Each column may also be compressed with a different method; the most significant compression for performance optimization is on attributes that are used in table ordering, i.e. in *ORDER BY* clause. Performance improvement mainly involves reduction of spent time in Input/Output operations. For columns of two possible values only, Run-length encoding is an appropriate method, since it stores only a number of occurrences of each value.[7]

Summary

Benefits of column-oriented database systems are fulfilled mainly for read-only queries, since they directly reach columns that participate in the query and do not operate on redundant data. Especially, column-oriented database systems are useful when a vast number of rows and a small number of columns are processed. Row-oriented database systems are efficient mainly for inserting, updating and deleting operations, since they work with one entity as with one file. Row-oriented database systems are useful mainly when a small number of rows and a vast number of columns are processed.

3 Vertica

This chapter introduces the Vertica database system in detail. It starts with definitions of basic terminology and continues with the description of Vertica is Sections 3.2 – 3.4.

3.1 Glossary

This section provides an overview of basic definitions of terms used in this thesis where are further explained in appropriate context. Besides, it is assumed that the reader is familiar with SQL language and principles of relational database systems.

CARDINALITY
Number of unique values in an attribute. Vertica defines three different types of cardinality[8][1]:

- *high cardinality* – high number of unique values, close to the number of records/rows, e.g. identifiers,
- *normal cardinality* – fewer unique values, e.g. first names,
- *low cardinality* – low number of unique values, e.g. sex.

CLUSTER TOPOLOGY
An object that maintains information about cluster topology – topology of particular nodes in Vertica database management system (DBMS).[2]

CATALOG RESPONSE TIME
Time of catalog response during checking query syntax. The check occurs after submitting of any query.

DATABASE CATALOG
Database catalog[8] is considered as a set of particular files in a database that hold metadata about database objects; for example tables, entities, projections, users and constraints.[3]

1. In Vertica documentation, chapter Cardinality
2. Taken from an email conversation with Vertica support (Savyuk, Pavel).
3. In Vertica documentation, chapter Catalog and Data Files

DESIGN
Set of projections definition for given tables.

DESIGNING PROCESS
The process of Database Designer tool that includes especially (1) adding query workload and tables, (2) setting goals and objectives, and (3) creating new projections. The process starts with the initial design proposal and ends with a design dropping.

DESIGNING STEP
Execution of Database Designer function *DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY* that directly creates new projections.

GLOBAL CATALOG
An object that maintains meta-data across all nodes in Vertica DBMS.[9]

LOCAL CATALOG
An object that maintains meta-data that refers to a local node in Vertica DBMS.[9]

LOGICAL AND PHYSICAL SCHEMA
Logical schema[8] of a database is defined as a set of tables, whereas physical schema of a database is defined as a set of projections.[4]

RESPONSE TIME
Duration of query execution that begins with query call and ends with returning query result.

SCHEMA
The term *schema* refers to a namespace in database where tables and projections are situated. Vertica enables users to create own schemata too.

4. In Vertica documentation, chapter Schema

3.2 Characteristics

Vertica is the only one database system that is both fully column-oriented and massive parallel processing (MPP). "Fully column-oriented" means that Vertica does not use features of row-oriented database system like indexes and use only features typical for column-oriented database systems.[5] MPP signifies that Vertica operates across multiple nodes simultaneously. Furthermore, it means that Vertica provides no single point of failure[6], and that resource scaling is rather horizontal in the form of adding new nodes.[10]

Vertica was built on the open-source project named C-Store in 2005 by Michael Stonebraker who was also one of the founders of the C-Store project. The C-Store project is described in detail in [11]. In 2011, Vertica was bought by Hewlett-Packard.[12] To April 2017, the current version of Vertica is 8.0.1 and was released in December 2016.[13]

This chapter provides an introduction to Vertica database system and its significant features. Precise descriptions of particular features are defined in [12], [14] and [15], and in the official documentation of Vertica [8].

Vertica is designed, as well as all column-oriented database systems, for analytic purposes. It means that transactional workloads, like data insertion and modification, are not efficient in column-oriented database systems in comparison to row-oriented database systems.[12] Vertica development is described in [12] as follows:

> *"Vertica was written entirely from scratch with the following exceptions, which are based on the PostgreSQL (...) implementation:*
>
> 1. *The SQL parser, semantic analyzer, and standard SQL rewrites.*
> 2. *Early versions of the standard client libraries, such as JDBC and ODBC and the command line interface.*
>
> *All other components were custom written from the ground up. While this choice required significant engineering effort and delayed the initial introduction of Vertica to the market, it means Vertica is positioned to take full advantage of its architecture."*

5. Taken from an email conversation with Vertica support (Savyuk, Pavel).
6. Shared-nothing architecture

3.2.1 Projections

In Vertica, projections represent physical schema of database and are one of its most characteristic features. In the same way, as in row-oriented database systems, a logical schema is represented by tables. Nevertheless, tables in Vertica do not include data; it is stored actually in projections. Each projection consists of a subset of table columns; it means that projections do not have to contain all columns of given tables. Tables may include an unlimited number of projections; however, each table needs to include at least one so-called *superprojection* that is described further on page 15. Benefits of multiple projections include higher query evaluation performance, since particular projections may be optimized for a specific query workload. For that purpose, particular projections of one table may be ordered and segmented by different columns. Moreover, columns of any table may have different encoding across all projections. On the other hand, multiple projections may also negatively affect database performance, since each additional projection means new duplicity in database. These duplicities cause that more storage is used and more resources are required for operations that change the content of tables, since all projections must be consistent.[12]

The example of the table with three projections is shown in Figure 3.1. The table defines the logical schema, and the projections store data. Each projection is ordered in a different way; moreover, the *customer_projection3* projection contains only two columns. The particular projections might be optimized for these queries:

- *customer_projection1*:

  ```
  SELECT c1, c2, c3 FROM customer_table WHERE c1 = 721
  ```

- *customer_projection2*:

  ```
  SELECT c2, c1, c3 FROM customer_table WHERE c2 = True
  AND c1 > 50 ORDER BY c2
  ```

- *customer_projection3*:

  ```
  SELECT c3, SUM(c1) FROM customer_table GROUP BY c3
  ```

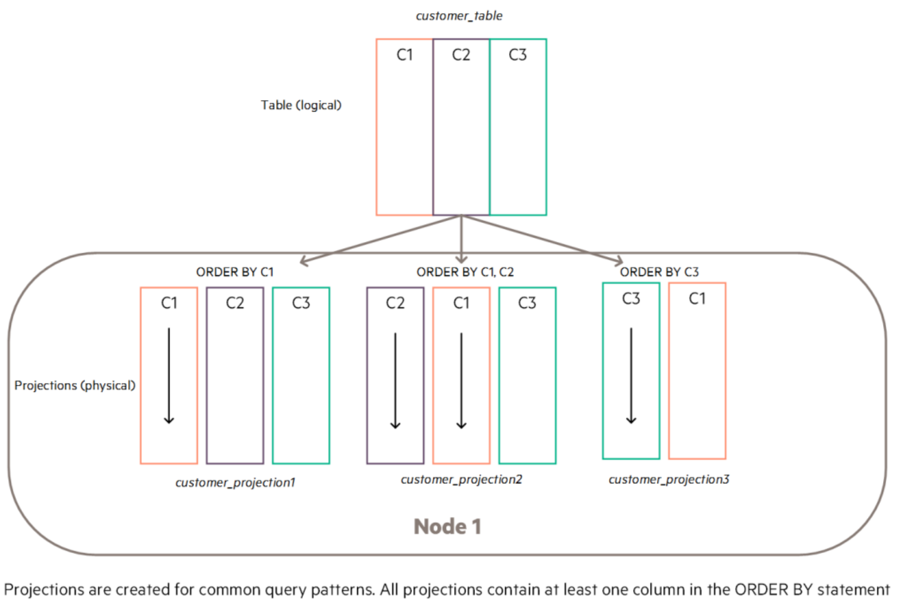


Figure 3.1: Principle of projections in Vertica.[16]

The command to create a projection with relevance to the table in Figure 3.1 is shown below:

```
CREATE PROJECTION customer_projection1
    (C1 ENCODING AUTO,
    C2 ENCODING DELTAVAL,
    C3 ENCODING RLE)
AS
    SELECT C1, C2, C3
    FROM customer_table
    ORDER BY C1
```

Vertica supports several different types of projections that are listed below[8, 16][7]:

- *Superprojection* – Projection that contains all columns from logical schema of a given table. Every table must include at least one superprojection to ensure availability of all data. Superprojection is automatically created with the initial data insertion to a ta-

7. In Vertica documentation, chapter Types of Projections

ble. When a new column is added to a table, the column is created only in superprojections. The other projections remain untouched. Furthermore, superprojection is also used for copying data into a new projection. No superprojection is specified in Figure 3.1, but the projections *customer_projection1* and *customer_projection2* contain all columns of tables *customer_table*, so they are actually superprojections.

- *Query-specific projection* – Projection that is optimized for a specific query workload and contains only needed columns.
- *Buddy projection* – Copy of an existing projection situated in a different node to assure data availability. Buddy projections must be identical to the projection it originates.
- *Live aggregate projection* – Projection that includes data aggregated from a source table[8]. It reduces aggregation operations on the source table.
- *Pre-join projection* – Projection that is a result of the *inner join* operation between two tables that are related with primary-key and foreign-key constraints.
- *Top-K projection* – Projection that contains only top-k rows from results of given query.
- *Projection with expressions* – Values of any columns are calculated from an anchor projection.

Vertica includes a built-in tool that designs new projections according to an input query workload. This tool, which optimizes physical schema of a database, is named Database Designer and is described in Section 4.3.

8. A source table for *Live aggregate projection*, *Pre-join projection*, *Top-K projection* and *Projection with expressions* is called *Anchor table* in Vertica.[8] – In Vertica documentation, chapter Anchor Table

3.2.2 Encoding

Vertica uses many types of column encodings; examples of them are listed below[8, 12][9]:

- *Auto* – Default encoding that is automatically chosen by Vertica in a case of an insufficient amount of representative data and in a case of table creation if not specified otherwise.
- *RLE (Run Lenght Encoding)* – Sequences of identical values are encoded as pairs of values and a number of occurrences. RLE is ideal for low cardinality and sorted columns.
- *Delta Value* – Values are stored as differences from the smallest value in a data block. It is intended for many-valued and unsorted columns.
- *Compressed Delta Range* – Values are stored as differences from a previous value. It is ideal for many-valued, numerical and sorted columns.
- *Block Dictionary* – Values are stored in a dictionary and columns contain only references to this dictionary. It is intended for few-valued and unsorted columns.

3.2.3 Segmentation

Vertica provides two types of data distribution across the whole cluster, data *replication* and *segmentation*. The first one, replication, keeps a copy of given projection in each node of a cluster. The second one, segmentation, distributes data of given projection among particular nodes. It means that cluster does not contain any copy of the data in different nodes and so the data is available only on one node. Segmentation is determined with *... SEGMENTED BY expression ...* clause in *CREATE PROJECTION* definition. The *expression* statement is replaced with a segmentation function and columns. Furthermore, columns used in that function should have high cardinality and even value distribution to split data equally over all nodes. The most common column used for segmentation is the primary key.[12] The example of segmentation is shown in Figure 3.2 where the column with identification numbers is segmented between three nodes equally with the usage of the function *"(id - 1 mod 3) + 1"*.

9. In Vertica documentation, chapter Encoding-Type

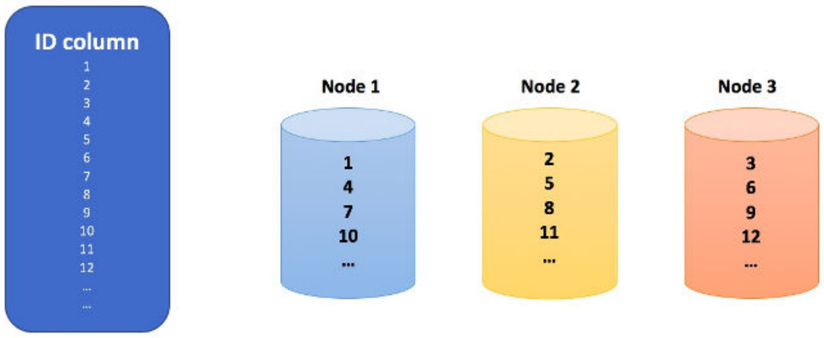


Figure 3.2: Principle of data segmentation in Vertica.

3.2.4 Partitioning

Segmentation defines data distribution among all nodes in a cluster. Moreover, Vertica enables also data distribution within a single node. This approach is called *partitioning*[12] and separates disk structures into particular logical regions. Partitioning is determined with *... PARTITION BY expression ...* clause in *CREATE PROJECTION* definition. The first benefit of partitioning is that query may be performed on different logical regions in parallel. The second benefit is that data may be partitioned by a specific pattern. For example, data may be partitioned by month extracted from a date column. Then each partition (logical part) contains data for one month as it is shown in an example in Figure 3.3.

3.2.5 High Availability

In Vertica, high availability of data is ensured in a cluster with K-Safety setup that indicates a number of replications of projections; these replications are called *buddies*. In a case of node failure, K-Safety ensures that data is still available on another node. *K*-value determines a number of nodes that may fail; possible values in Vertica are only *0*, *1* and *2*. Independently, if a half or more nodes fail, database is stated as *unsafe* and is shut down. The exemplary situation in a cluster with K-Safety 2 is shown in Figure 3.4. There, in spite of failures of Nodes 2 and 3, database is still running, and all data is available. If Node 5 fails, despite the fact that all data is still available, the database would be shut down, because a number of failed nodes is higher than the half of all nodes. [8][10]

10. In Vertica documentation, chapter K-Safety

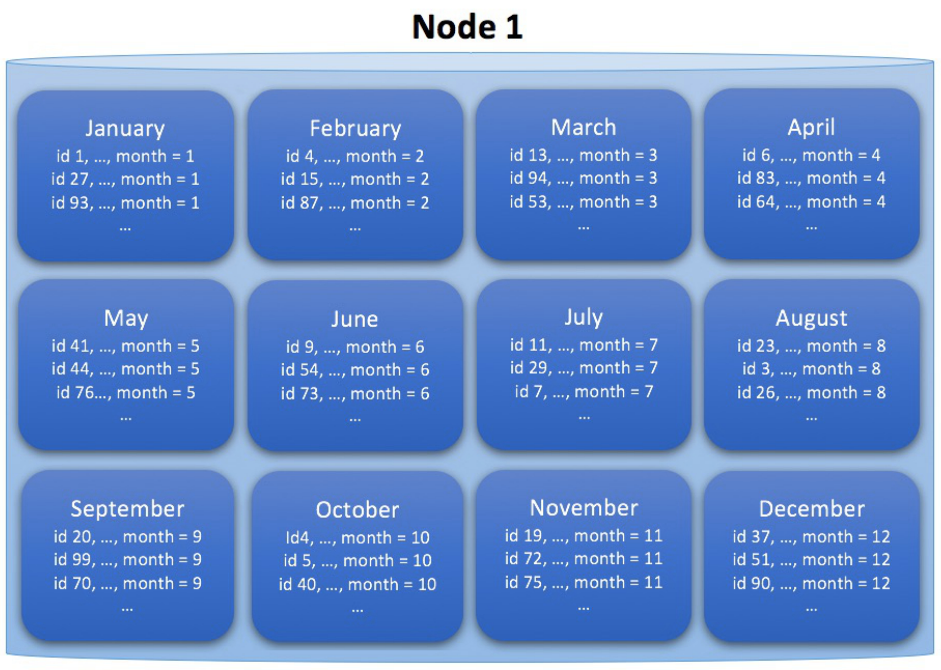


Figure 3.3: Principle of data partitioning in Vertica. The data is partitioned according to the column with months.

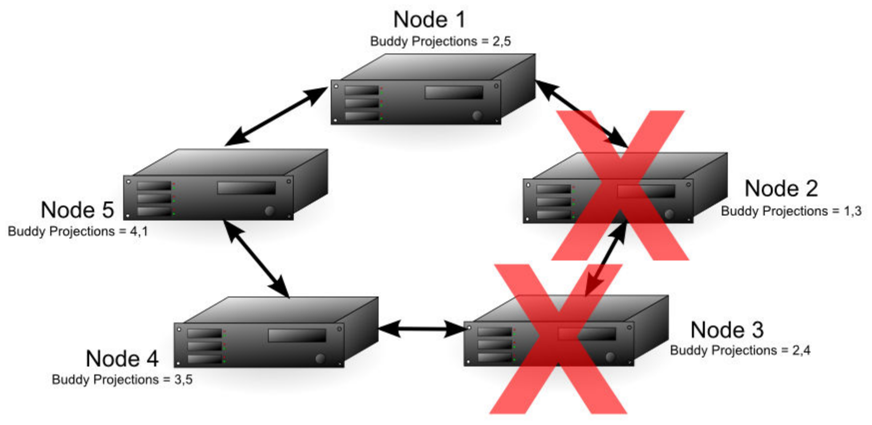


Figure 3.4: Exemplary usage of K-Safety with value 2.[8]

3.3 Locks

Vertica ensures concurrency and consistency of data with locks. This approach is essential since multiple users may access database simultaneously. Moreover, multiple transactions of one user may require the same data. For that reason, Vertica must ensure that these transactions manipulate with consistent data.

Vertica includes two types of locks. *Local catalog* locks and *global catalog* locks are examples of *system locks*. Locks on particular objects in database (e.g. projections, tables) are parts of *object locks*. Besides, locks in Vertica are defined with several modes that are described in Appendix A.[8, 12][11]

3.4 System tables

Vertica provides system tables that gather information about system's resources, background processes, workload, and performance. Furthermore, system tables allow a user to perform more detail query profiling, data diagnosing and viewing historical data. Selected system tables used further in this thesis are listed in Table 3.1. System tables are separated into three schemata:

- *v_catalog* – Contains tables with information about objects residing in database; for example tables, projections, resource pools, and users.
- *v_monitor* – Contains tables with information about the current situation in database; for example configuration parameters, locks and memory usage.[17]
- *v_internal* – Contains tables with information about current events in database; for example important system and user activities. These system tables are called *Data collectors* and have prefix *dc_*.[8][12]

Gathering information into system tables is automatically turned on. The database administrator may change many setting of this gathering; for example, they may change retention time of data in system tables and maximum size of particular system tables.[17]

11. In Vertica documentation, chapter Locks
12. In Vertica documentation, chapter Data Collector

System tables	**Description**
execution_engine_profiles	Provides profiling information about query execution runs.
query_requests	Returns information about user-issued query requests.
dc_requests_issued	History of all SQL requests issued.
dc_requests_completed	History of all SQL requests completed.
dc_resource_acquisitions	History of all resource acquisitions.
dc_resource_releases	History of all resource acquisition releases.
dc_lock_attempts	History of lock attempts (resolved requests).
dc_lock_releases	History of lock releases.
dc_explain_plans	Explain plans.

Table 3.1: Selected system tables that are used in this thesis. Descriptions of *execution_engine_profiles* and *query_requests* are taken from the official documentation [8]. Descriptions of data collectors are taken from the system table *v_monitor.data_collector*.

4 Database tuning

Here, we provide database tuning principles used in relational databases that are also applicable in Vertica. Database tuning should not be skipped since it may save a huge number of resources, and database may run much faster.

4.1 General database tuning

This section provides an overview of general tuning principles in relational databases. Usage of indexes and its tuning have a great influence on database performance. Nevertheless, column-oriented database systems typically do not use indexes, and for that reason, tuning principles of indexes are not described.

General principles of tuning in relational databases are described below in the alphabetical order:

AGGREGATE MAINTENANCE

In a case of frequent execution of aggregation operations, creating auxiliary tables with aggregated data helps increase database performance. Then, aggregation functions are not performed repeatably, but the result is directly read from the aggregated auxiliary table. This approach requires updates of auxiliary tables with every update of primary tables. Thus, benefits of tables with aggregated data must exceed additional costs of updating data in that tables.[18]

DENORMALIZATION

Normalization of database is used in a designing process of a logical schema, since it assures that data is stored into a correct table. If database schema is normalized, redundancy of data is minimized, and many possible inconsistencies that may occur within a work with data are eliminated as well. Moreover, tables should be controlled that they contain a minimum of functional dependencies among their columns.[18, 19]

Normalization typically requires a database schema to fulfil the 3rd normal form. The normal forms are[19]:

- *First Normal Form* – Attributes contain only atomic values and no duplicated records (rows).

- *Second Normal Form* – All non-primary-key attributes depend on the whole primary key. This normal form allows a transitive dependency on the key.

- *Third Normal Form* – All non-primary-key attributes depend directly on the primary key, i.e. attributes do not involve any transitive dependency.

Denormalization is a reverse process of normalization; it violates constraints that assure normalization of tables. Denormalization may increase performance only in specific situations when a table is rarely updated. In that case, non-update queries are faster on denormalized tables, because data is situated in fewer tables and the queries do not need to perform join operations. For example, denormalization may be used for archiving data since no update is expected on that data. [18]

ELIMINATION OF FOREIGN KEY CONSTRAINTS

Foreign key constraints negatively affect database performance, since they must be controlled whenever data is modified or inserted. For the purpose of better performance, the control of foreign keys constraints may be moved to an application layer if that application allows it.[20]

EXPLAIN COMMAND

EXPLAIN command serves for monitoring of query execution. In this thesis, the monitoring with EXPLAIN command is described in Section 4.2.1.[20]

HINTS

Hints in database queries may enforce specific operations during their executions. They may positively affect query performance and provide required behavior of database.[20]

MATERIALIZED VIEW

Materialized View, as well as the *Aggregate Maintenance*, servers for eliminating costs in a case of frequent queries. In general, materialized views contain results of a specific query and also requires additional costs for updating data.[18]

STATISTICS
Updated statistics of each column increase performance, since a database has an overview of data distribution and may modify the strategy of evaluating a query from index scan to sequential when many rows match the query predicate, for example.[20]

4.2 Database tuning in Vertica

Projections may be tuned in two different ways. In the first case, Vertica enables using built-in tool that automatically designs projections on the basis of a query workload. This tool is described in Section 4.3. In the second case, projections may be designed manually on the basis of optimization rules and best practices that are described in the official documentation [8]; for example in chapters *Optimizing Query Performance* and *Choosing Sort Order: Best Practices*. Principal optimizations are summarized in paragraphs below.[8][1]

CARDINALITY
The best practices recommend to include columns with low cardinality early in the projection sort order, especially if the low-cardinality columns are combined with RLE. In that case, a projection size is minimized, and performance is optimized.

ENCODING
Examples of encoding were already presented in Section 3.2.2. Precise usage for each encoding type is described in chapter *Encoding-Type* in the official documentation [8].

EQUAL CARDINALITY
If two columns have the same cardinality, it is recommended to include the column with a higher number of records earlier in the projection sort order than the column with a lower amount of records.

1. In Vertica documentation, chapters Choosing Sort Order: Best Practices; and Optimizing Query Performance

GROUP BY

Two types of *GROUP BY* operations are involved in Vertica:

- *GROUP BY PIPELINED* – Grouped columns are pre-sorted. No pre-processing of data is performed. This type is more efficient than *Group by hash* since it uses less memory and time.
- *GROUP BY HASH* – Grouped columns are not pre-sorted. For that reason, pre-processing of data is required.

The performance of both types is similar when tables are small. Chapter *Optimizing Query Performance* in the official documentations [8] involves three additional conditions when *Group by pipelined* operation is performed. One example of this condition is that *Group by pipelined* is used when all columns from *GROUP BY* clause are present in the projection's sort order.

IMPORTANT QUERIES

If any query is performed more frequently than the others, it is efficient to optimize projections for this query (e.g. predicates, *GROUP BY* columns).

JOIN

Two types of joins are involved in Vertica:

- *Merge join* – Join columns are pre-sorted. No pre-processing of data is performed. This type is more efficient than *Hash join* since it uses less memory and time.
- *Hash join* – At least one column is not pre-sorted. An in-memory hash table is created from the table with fewer records (inner table). Then, records from hash table are matched with records from a larger table. This type is not as efficient as *Merge join* since it needs additional memory for creating that hash table.

The best practices recommend to include join columns into the first positions of sort orders of both tables. This may be done in the form of creating new additional projections to the existing projections. *Merge join* is especially important when tables are so large that they cannot fit into memory. In comparison, if both tables fit into memory, a time difference between both joins is small.

PREDICATES

The best practices recommend to include columns that participate in predicates early in the projection sort order, especially if the columns are combined with low-cardinality. In that case, evaluation of predicate is more efficient, since the column is sorted and the query operates only on the defined part of that column.

SEGMENTATION

Resegmentation[2] of data occurs when a query contains *GROUP BY* clause and projections are not sorted according to the columns of this clause. To avoid data pre-processing (resegmentation), Vertica recommends todninclude columns from segmentation in *GROUP BY* clause.

4.2.1 Query monitoring

Query monitoring is a necessary step for increasing query performance. Vertica provides several possibilities of monitoring:

1. Query plan
2. Query profiling
3. System tables

QUERY PLAN

The term *query plan* is connected to the EXPLAIN command and its definition in the official documentation [8] in chapter *Query Plans*, is:

> *"A query plan is a sequence of step-like paths that the Vertica cost-based query optimizer uses to execute queries. Vertica can produce different query plans for a given query. For each query plan, the query optimizer evaluates the data to be queried: number of rows, column statistics such as number of distinct values (cardinality), distribution of data across nodes. It also evaluates available resources such as CPUs and network topology, and other environment factors. The query optimizer uses this information to develop several potential plans. It then compares plans and chooses one, generally the plan with the lowest cost."*

2. *"A process that Vertica performs automatically during query execution that distributes the rows of an existing projection or intermediate relation evenly to each node in the cluster. At the end of resegmentation, every row from the input relation is on exactly one node.."*[8] – In Vertica documentation, chapter Resegmentation

Query plan compares potential query paths according to costs that are defined in the official documentation [8], chapter *Query Plan Cost Estimation* as:

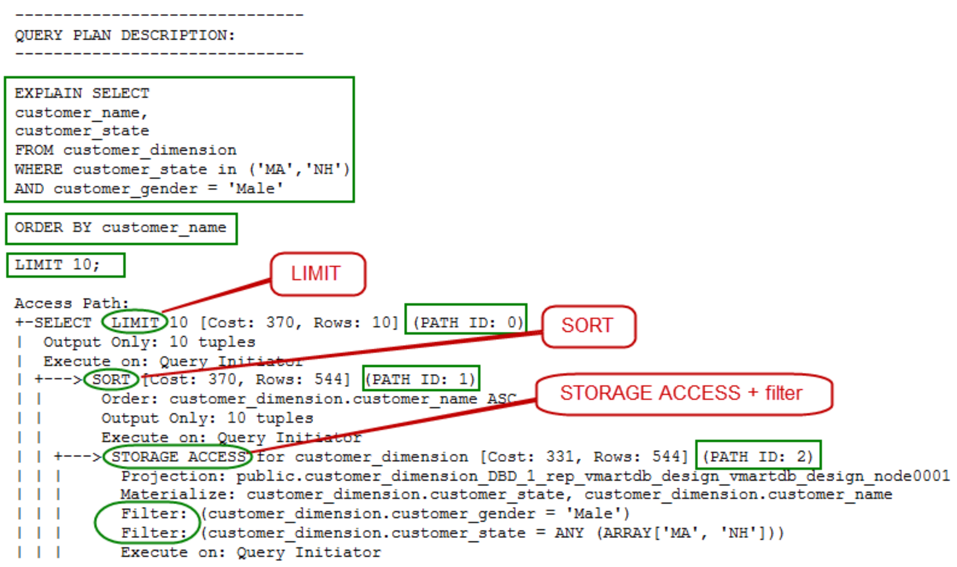
> *"The query optimizer chooses a query plan based on cost estimates. The query optimizer uses information from a number of sources to develop potential plans and determine their relative costs. These include:*
>
> - *Number of table rows;*
> - *Column statistics, including: number of distinct values (cardinality), minimum/maximum values, distribution of values, and disk space usage;*
> - *Access path that is likely to require fewest I/O operations, and lowest CPU, memory, and network usage;*
> - *Available eligible projections;*
> - *Join options: join types (merge versus hash joins), join order;*
> - *Query predicates;*
> - *Data segmentation across cluster nodes."*

The output of EXPLAIN command contains the most efficient query plan for given query with estimated costs for each operation. The EXPLAIN command is very useful for finding non-optimal steps of queries; for example, it shows a precise order of operations, and tables that participate in particular joins. Costs in query plans are only estimated, so system tables must be queried for precise values of used resources. An example of the query plan is shown in Figure 4.1.

The verbose clause of EXPLAIN causes more detailed information about resources to be output. Exemplary operator from EXPLAIN verbose command with used resources and their costs is shown below[22]:

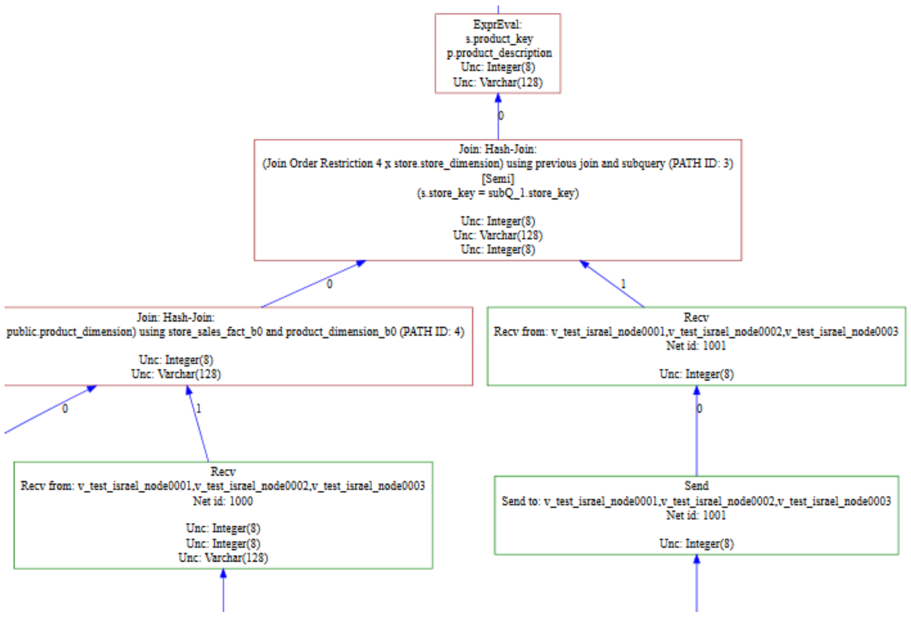
```
GROUPBY HASH (SORT OUTPUT) (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMENT GROUPS) [Cost: 2662115.000000, Rows: 15000000.000000
Disk(B): 2400000000.000000 CPU(B): 2280000000.000000 Memory(B): Network(B):
4440000000.000000 Parallelism: 3.000000] [OutRowSz (B): 144] (PATH ID: 2)
```

Both EXPLAIN command and EXPLAIN verbose command also contains output for a graphical demonstration of a query plan. This output is in the form of source for visualization tool Graphviz. Exemplary graph created from an output of EXPLAIN command is shown in Figure 4.2.



```
------------------------------
QUERY PLAN DESCRIPTION:
------------------------------

EXPLAIN SELECT
customer_name,
customer_state
FROM customer_dimension
WHERE customer_state in ('MA','NH')
AND customer_gender = 'Male'

ORDER BY customer_name

LIMIT 10;

Access Path:
+-SELECT  LIMIT 10 [Cost: 370, Rows: 10] (PATH ID: 0)
|  Output Only: 10 tuples
|  Execute on: Query Initiator
| +---> SORT [Cost: 370, Rows: 544] (PATH ID: 1)
| |      Order: customer_dimension.customer_name ASC
| |      Output Only: 10 tuples
| |      Execute on: Query Initiator
| | +---> STORAGE ACCESS for customer_dimension [Cost: 331, Rows: 544] (PATH ID: 2)
| | |      Projection: public.customer_dimension_DBD_1_rep_vmartdb_design_vmartdb_design_node0001
| | |      Materialize: customer_dimension.customer_state, customer_dimension.customer_name
| | |      Filter: (customer_dimension.customer_gender = 'Male')
| | |      Filter: (customer_dimension.customer_state = ANY (ARRAY['MA', 'NH']))
| | |      Execute on: Query Initiator
```

Figure 4.1: An example of Vertica query plan.[21]



ExprEval:
s.product_key
p.product_description
Unc: Integer(8)
Unc: Varchar(128)
0
Join: Hash-Join:
(Join Order Restriction 4 x store.store_dimension) using previous join and subquery (PATH ID: 3)
[Semi]
(s.store_key = subQ_1.store_key)
Unc: Integer(8)
Unc: Varchar(128)
Unc: Integer(8)
0
1
Join: Hash-Join:
public.product_dimension) using store_sales_fact_b0 and product_dimension_b0 (PATH ID: 4)
Unc: Integer(8)
Unc: Varchar(128)
Recv
Recv from: v_test_israel_node0001,v_test_israel_node0002,v_test_israel_node0003
Net id: 1001
Unc: Integer(8)
0
1
0
Recv
Recv from: v_test_israel_node0001,v_test_israel_node0002,v_test_israel_node0003
Net id: 1000
Unc: Integer(8)
Unc: Integer(8)
Unc: Varchar(128)
Send
Send to: v_test_israel_node0001,v_test_israel_node0002,v_test_israel_node0003
Net id: 1001
Unc: Integer(8)

Figure 4.2: An example of a graph that was created from an output of EXPLAIN command. The figure contains only a part of the whole graph.[22]

QUERY PROFILING
System tables in Vertica include the table *QUERY_PLAN_PROFILES* that contains real-time information about a run of given query. For each query operation, this information involves execution time, spent memory, a size of data that went through the network and whether the operation was completed.[8][3]

SYSTEM TABLES
Vertica maintains many useful system tables that provide a huge amount of information; selection of specific system tables depends only on user's monitoring requirements.

4.3 Database Designer

Vertica includes a built-in tool that optimizes physical schema of database, and that is named *Database Designer*. The tool should save costs needed for manual optimization. The physical schema is optimized according to a query workload. The tool is managed with separate functions that determine particular steps of the designing process and that are described in Section 4.3.1.

The output of Database Designer includes two scripts. The first script contains definitions of newly designed projections. Besides these definitions, the second script contains also queries for dropping old projections and refreshing tables. For that reason, the second script is used for deployment of a new design.[15]

4.3.1 Functions of tool

This section includes basic functions that are used within Database Designer. All functions are described in the official documentation [8][4]. The basic functions are:

DESIGNER_CREATE_DESIGN
The step that initializes a designing process. An object with a design is created in system tables. Besides, Database Designer creates three schemata that contain auxiliary tables for a designing step.

3. In Vertica documentation, chapter Query Plan Profiles
4. In Vertica documentation, chapter Database Designer Functions

DESIGNER_ADD_DESIGN_TABLES
Tables referenced in a query workload are added to a design.

DESIGNER_ADD_DESIGN_QUERY
Queries are added to a design. Particular queries may have different weight in a design (the highest weight is 1, the lowest weight is 0); a default weight is set to 1.

DESIGNER_SET_DESIGN_TYPE
Database Designer enables choosing one of two design types that specify a form of designed projections. These types are:

- *Comprehensive* – This type redesigns projections for all added tables. New design contains new projection definitions for all tables. During a deployment of new projections, old projections are dropped. This type is more suitable for newly created or fulfilled tables, and for the first run of Database Designer.
- *Incremental* – This type creates new additional projections while the old projections are retained. The type is more suitable for a situation when new queries are added to an existing query workload.

DESIGNER_SET_OPTIMIZATION_OBJECTIVE
Database Designer enables setting an optimization objective that specifies whether the creation of projection should focus on a small footprint or high performance. Possible objectives are:

- *Query* – This objective is focused on high performance of query workload. For that reason, more projections are typically created.
- *Load* – This objective is focused on footprint to ensure small the size of data. With this objective, queries may be slower than with *Query* objective.
- *Balanced* – Balance between objectives *Query* and *Load*.

DESIGNER_SET_DESIGN_KSAFETY
This function sets a K-Safety of designed projections.

DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY
This function directly creates new projections that are optimized for the added query workload. The function is described in Section 4.3.2.

DESIGNER_DROP_DESIGN
This function drops a design from database (system tables) together with three schemata that contain auxiliary tables.

4.3.2 Designing step

The main function of Database Designer that directly runs the design execution is *DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY*. The whole process of designing new projections is shown in Figure 4.3. In a nut shell, individual steps implementing the function can be divided into two phases: *Query Optimization Phase* and *Storage Optimization Phase*. The precise algorithm of design creation is described in detail in [15]. Parameters of this function are described in Appendix B.

Query Optimization Phase affects query performance much more than *Storage Optimization Phase*. The steps of *Query Optimization Phase* are:

1. Parsing queries and their association with input tables.
2. Enumeration of projection's orderings and their comparison. The comparison of projection candidates is performed with a usage of costs that are computed from input/output costs, network costs, memory costs, CPU costs and parallelism costs.
3. Enumeration of projection's segmentations and their comparison. Besides, Database Designer considers whether to use replication of data on all nodes instead.
4. Generation of candidates for projection.
5. After the phase of cost optimizer, the most optimal projection candidate is advanced to the next phase.

Storage Optimization Phase affects load performance and storage. The main part of this phase involves choosing encodings for each column. This process is very intensive for resources and time but still very effective, since the encoding directly affects load performance and size of the footprint. For each column, encoding candidates are generated. These candidates include all encoding types except those that have been excluded

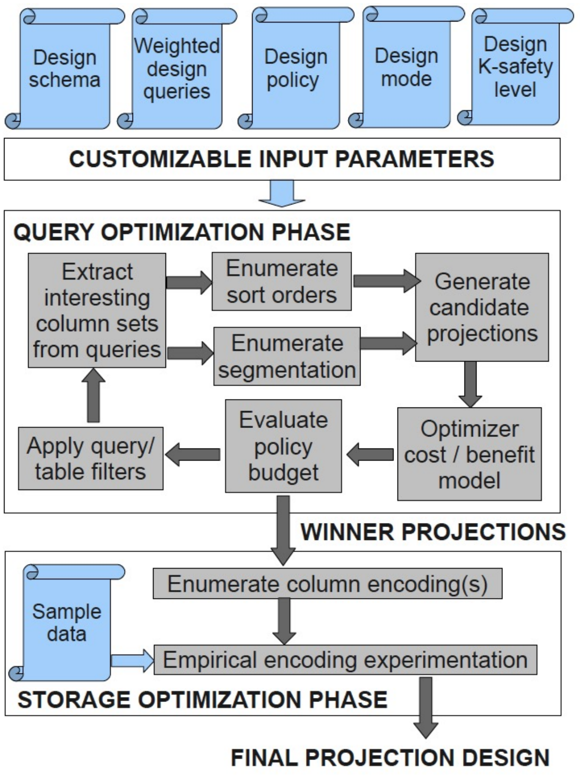

Design schema
Weighted design queries
Design policy
Design mode
Design K-safety level
CUSTOMIZABLE INPUT PARAMETERS
QUERY OPTIMIZATION PHASE
Extract interesting column sets from queries
Enumerate sort orders
Enumerate segmentation
Generate candidate projections
Optimizer cost / benefit model
Evaluate policy budget
Apply query/ table filters
WINNER PROJECTIONS
Sample data
Enumerate column encoding(s)
Empirical encoding experimentation
STORAGE OPTIMIZATION PHASE
FINAL PROJECTION DESIGN

Figure 4.3: Process of designing new projections with Database Designer.[15]

by Database Designer. The excluding process is performed with a usage of data type, cardinality, and sortedness of given column; this information helps define encoding types that are surely non-optimal for that column. Then, Database Designer extracts a sample of column data and compares all candidates by compressing this sample. So, the candidates with minimal footprint are identified. The second step of *Storage Optimization Phase* is selecting appropriate column set.[15]

5 DBPerfComp tool

The DBPerfComp tool was written in Python and enables several modes of its usage. The main mode named *Test Design* provides an automatic test that analyzes designs created by Database Designer. The Use case diagram of the DBPerfComp tool is presented in Figure 5.1. Descriptions of individual modes are below:

- *Run queries* – Running input queries on input schemata and storing measured data to a database.
- *Copy schema* – Creating a new schema (without data) and copying tables from an old schema to the new schema.
- *Create schema* – Creating a schema with an input schema definition, data path, and path to the *copy* statements.
- *Deployment* – Deploying new projections with an input definition script.
- *Design* – Designing new projections with the Database Designer.
- *Test Design* – Complex testing mode that searches for the worst cases among the designs of Database Designer.

5.1 Test goal

The tool targets weaknesses of the Database Designer in the form of non-optimally designed projections. The optimality of projections is determined with a comparison of projections designed by Database Designer and projection not designed by that tool (called *base schema*). The comparison is done with the usage of an average response time of query workload. If the response time is higher with projections designed by Database Designer, the query workload is further investigated by adding a new query. Furthermore, the test investigates an influence of the different amount of information (referential integrity, segmentation) in table definition (*CREATE TABLE* statement) on the optimality of projections. The deep manual analysis is subsequently performed on the measured data and also on the data from the system tables (see Chapter 7).

Within the analysis, similarities among the worst designs of the Database Designer are searched, and the features (e.g. specific joins, group by, ordering) that cause non-optimal designs are specified.

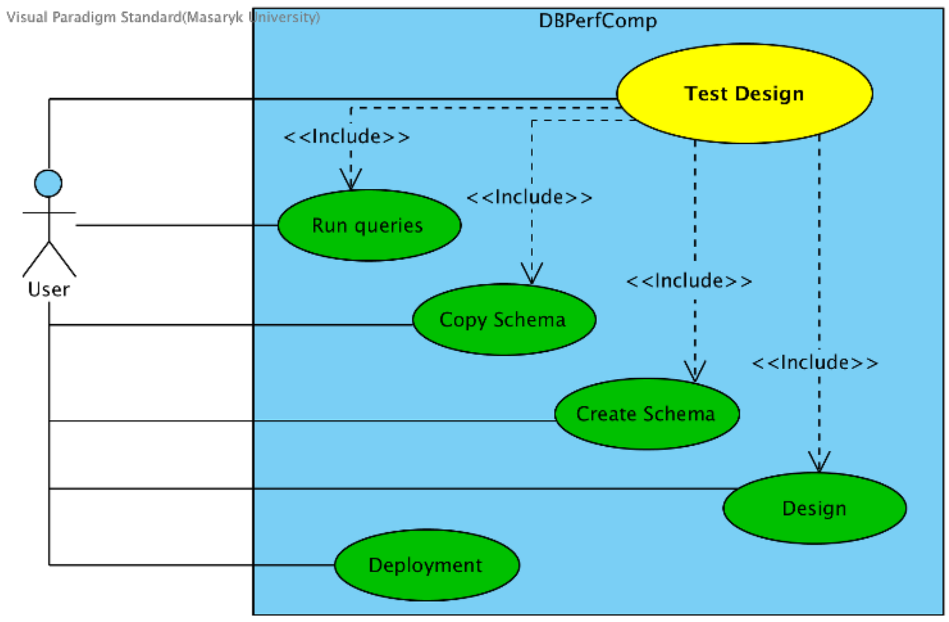


Figure 5.1: Use case diagram of the DBPerfComp tool

Since the tool is automatic, it may be used with every new release of the Vertica database system to test new features and to find new possible issues. Besides, the companies that deploy Vertica database may use this tool to confirm whether the Database Designer is eligible to be used by their customers or it negatively affects the performance of their production systems.

5.2 Tool design

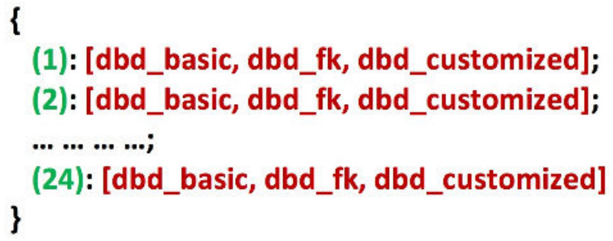
The DBPerfComp is implemented as the utility to test the performance of que-ries, so a user of it must specify several input parameters, mainly queries to analyze, a list of tables and their schemata. All configuration parameters are explained in Section 5.4.

QUERY BUCKET

Based on input queries, the tool creates *query workloads*[1] as their combinations, i.e. the power set of the set of queries. Furthermore, the tool works with the structure *query bucket* that stores pairs of the query

1. In the implementation, the term *query set* is used instead of the *query workload*.

workload and a list of schemata to test. The query bucket has actually a key-value structure (map) where the key is the query workload, and the value is the list of schemata. In the initial phase of the query bucket, each query from the configuration file is a separate query workload, and all schemata from the configuration file are bound to each query workload. For each query workload and tested schema, the tool creates a new copy of tested schema and uses Database Designer to design and deploy new projections. Figure 5.2 contains the query bucket with exemplary values. In Figures 5.2 – 5.6, query workloads are in the green, and they are formed as a set of query IDs; tested schemata are in the red.



```
{
  (1): [dbd_basic, dbd_fk, dbd_customized];
  (2): [dbd_basic, dbd_fk, dbd_customized];
  ... ... ... ...;
  (24): [dbd_basic, dbd_fk, dbd_customized]
}
```

Figure 5.2: Query bucket after DBPerfComp tool initialization – each workload contains one query.

ALGORITHMS

Pseudocode implementing the *Test design* mode is shown in Algorithm 1. From the code, two specific functions to run queries and to add new workloads to the query buckets are called. They are specified in Algorithms 2 and 3.

The algorithm starts with parsing the configuration file and initializing the query bucket with all queries as individual workloads. Next, the algorithms inspects each pair of workload and schema in the bucket as follows:

1. If the tested schema is not the base schema, the schema is created in the database as a copy of the input schema.
2. If the tested schema is not the base schema, Database Designer creates and deploys new projections.
3. Segmentations of projections are parsed.
4. Queries from the query workload are run on the tested schema and monitored.

5. If the tested schema is not the base schema, the query workload's ratio is computed as a ratio between the sum of response times of all queries that run on the tested schema and the sum of response times of all queries that run on the base schema.
6. If the ratio is higher than the threshold predefined in the configuration file, pairs of new query workloads and the tested schema are added to the query bucket. The new query workloads are created as combinations of tested query workload and every particular input query from the configuration file. In this step, the tool also checks for possible duplicities in the query bucket.
7. Data is stored in a database, i.e. measured data, system information, and query workload ratio.
8. The tested schema is dropped.

As a result, the tool stores measured data into three separate database tables and creates snapshots of required system tables. The whole output is described in Section 5.3. The configuration file is described in Section 5.4.

ADDING QUERY WORKLOADS

The function of adding new query workloads to the query bucket, shown in Algorithm 3, may change a content of the query bucket in several possible ways. The assumption for the figures on the following pages is that the configuration file contains queries *1, 2, 3 ... 24* and schemata *dbd_basic*, *dbd_fk*, *dbd_customized*. Further in the figures, the items with a gray background signify query workloads and schemata added in the given step. The green keys in the figures are query workloads, and the red values are tested schemata.

The first option of changing the query bucket's content is that it does not include the tested query workload. For that reason, the combination of query workload and tested schema is added into the query bucket. This situation is shown in Figure 5.3.

Algorithm 1 Test design

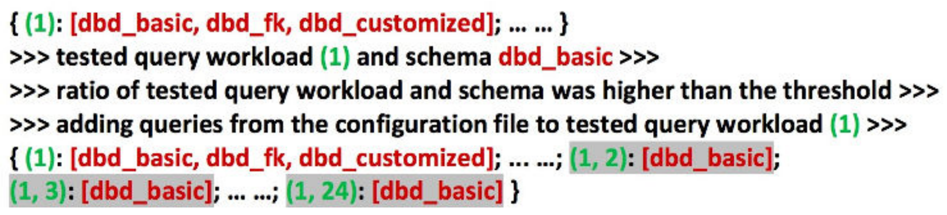
```
1: Base schema, Schemata, Queries, Tables ← Configuration file
2: Create Query bucket (dictionary)
3: for each Query in Queries do
4:     Query workload ← tuple(Query)
5:     Query bucket ← key: Query workload
6:     for each Schema in Schemata do
7:         Query bucket ← value: Schema set
8:     end for
9: end for
10: Create Query workload queue ← sorted keys of Query bucket    ▷ Query workload queue is described on Page 40
11: for each Query workload in Query workload queue do
12:     Create List schemata ← Base schema
13:     List schemata ← values in Query bucket [key: Query workload ]
14:     for each Schema in List schemata do
15:         if not Base schema then
16:             CREATE SCHEMA    ▷ With schema definition from the configuration file OR Copy tables from existing schema in database (see Documentation on Page 43)
17:             CREATE DESIGN WITH DATABASE DESIGNER    ▷ Running the Database Designer tool on the created schema with queries from the Query workload.
18:         end if
19:         PARSING SEGMENTATION    ▷ From all projections of tables from the configuration file.
20:         RUN_QUERY    ▷ Algorithm 2. Queries from the Query workload.
21:         MONITORING QUERIES    ▷ Storing data from each iteration of each query into the database.
22:         if not Base schema then
23:             Ratio ← (Sum of response times of all queries on Schema) / (Sum of response times of all queries on Base schema)
24:             if Ratio > Threshold then    ▷ Threshold from the configuration file.
25:                 ADD_QUERY_WORKLOAD(Query workload, Schema)    ▷ Algorithm 3.
26:             end if
27:         end if
28:         STORING MEASURED DATA INTO THE DATABASE
29:         if  not Base schema and All queries run successfully then
30:             DROPPING SCHEMA
31:         end if
32:     end for
33: end for
```

Algorithm 2 Running queries

```
1: function RUN_QUERY
2:     for each $Query\ in\ Query\ workload$ do
3:         for $iteration \leftarrow 1, 3$ do
4:             RUN QUERY(query, schema)
5:         end for
6:     end for
7:     Insert data into snapshot tables
8:     for each $Query\ in\ Query\ workload$ do
9:         MONITOR QUERY(query label)                ▷ From snapshot tables.
10:    end for
11: end function
```

Algorithm 3 Adding new query workloads to query bucket

```
1: function ADD_QUERY_WORKLOAD($Query\ workload, Schema$)
2:     for each $Query\ in\ Input\ Queries$ do ▷ Queries from the configuration file that are tested in the tool.
3:         if $Query$ not in $Query\ workload$ then
4:             $New\ query\ workload \leftarrow Query\ workload + Query$
5:             if $Length(New\ query\ workload) \leq Max\ set\ depth$ then ▷ Max set death from the configuration file.
6:                 if $Duplicate\ query\ workload$ in $Query\ bucket$ then
7:                     $Duplicity \leftarrow True$
8:                 end if
9:                 if $Duplicity = True$ and $Schema$ not in $Duplicity's\ schemata$ then
10:                    $Duplicity's\ schemata \leftarrow Schema$
11:                end if
12:                if $Duplicity = False$ then
13:                    $Add\ New\ query\ workload$
14:                end if
15:            end if
16:        end if
17:    end for
18: end function
```

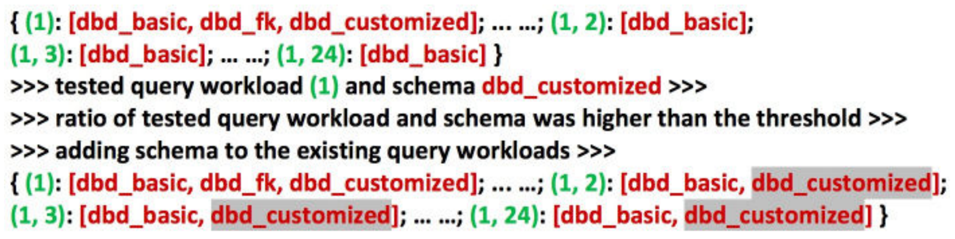


```
{ (1): [dbd_basic, dbd_fk, dbd_customized]; … … }
>>> tested query workload (1) and schema dbd_basic >>>
>>> ratio of tested query workload and schema was higher than the threshold >>>
>>> adding queries from the configuration file to tested query workload (1) >>>
{ (1): [dbd_basic, dbd_fk, dbd_customized]; … …; (1, 2): [dbd_basic];
(1, 3): [dbd_basic]; … …; (1, 24): [dbd_basic] }
```

Figure 5.3: Adding a new query workload (with tested schema) to the query bucket. The actually added items are in the gray background.

DUPLICITIES

The second option of changing the query bucket's content is that it already contains this query workload, but the tested schema is not included in its value. For that reason, the tested schema is appended to the list in the corresponding value. This situation is shown in Figure 5.4.



```
{ (1): [dbd_basic, dbd_fk, dbd_customized]; … …; (1, 2): [dbd_basic];
(1, 3): [dbd_basic]; … …; (1, 24): [dbd_basic] }
>>> tested query workload (1) and schema dbd_customized >>>
>>> ratio of tested query workload and schema was higher than the threshold >>>
>>> adding schema to the existing query workloads >>>
{ (1): [dbd_basic, dbd_fk, dbd_customized]; … …; (1, 2): [dbd_basic, dbd_customized];
(1, 3): [dbd_basic, dbd_customized]; … …; (1, 24): [dbd_basic, dbd_customized] }
```

Figure 5.4: Adding a schema (in gray background) to the existing query workload in the query bucket.

The query workload is stored as a tuple of queries. Therefore, the DBPerfComp tool controls possible duplicities in the query bucket, because the query workload (a, b, c) is the same as (b, c, a) for Database Designer, as well as for DBPerfComp tool. The data structure of the query bucket is a forest[2], where separate rooted trees[3] contains (key-value) combinations of query workloads and list of schemata. The root of each tree includes query workload with one query from the configuration file and a list of all schemata from the configuration file. Each tree expands with development of new query workloads (from the query workload of its root)

2. Forest is: *"Graphs containing no simple circuits that are not necessarily connected"*. Each connected component in a forest is a tree.[23]

3. *"A tree is a connected undirected graph with no simple circuits. (...) A rooted tree is a tree in which one vertex has been designated as the root and every edge is directed away from the root."*[23]

and bounding tested schemata to them. Since DBPerfTool creates duplicities of query workloads, branches are progressively restricted to eliminate them. The forest structure and the elimination of duplicities are presented in Figure 5.5.

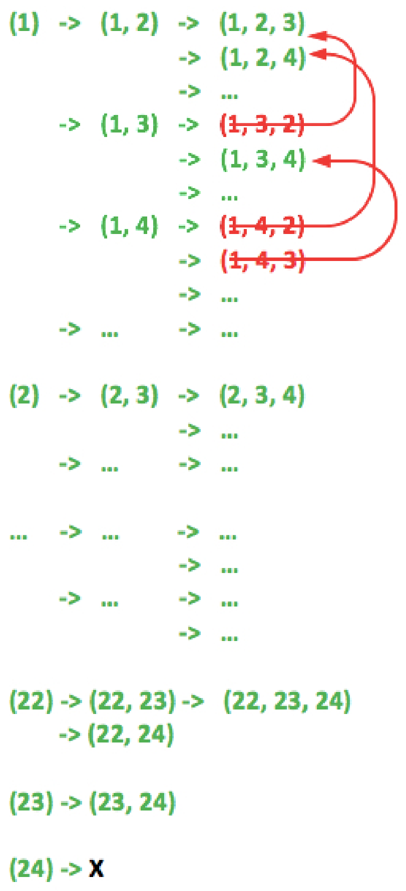
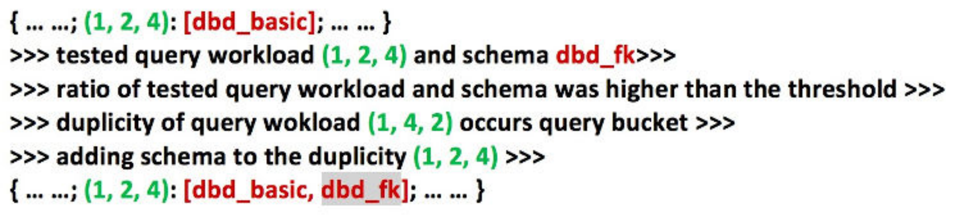


Figure 5.5: Query bucket has a forest structure where single rooted trees include (key-value) combinations of query workloads and lists of schemata. The root of each tree includes the only query as a key and schemata as a value from the configuration file. This figure shows only the structure of keys in order to greater comprehension of the query bucket data structure. Query workloads in the red present workloads that are detected as duplicities and are eliminated for that reason.

The third option of changing the query bucket's content is that it includes duplicity of the query workload, but the duplicity does not contain the tested schema. This situation is shown in Figure 5.6. Because the query workloads with one query are tested first[4], the new schema cannot be added to the query workload that was already tested.

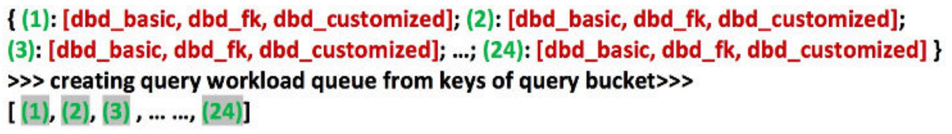


```
{ … …; (1, 2, 4): [dbd_basic]; … … }
>>> tested query workload (1, 2, 4) and schema dbd_fk>>>
>>> ratio of tested query workload and schema was higher than the threshold >>>
>>> duplicity of query wokload (1, 4, 2) occurs query bucket >>>
>>> adding schema to the duplicity (1, 2, 4) >>>
{ … …; (1, 2, 4): [dbd_basic, dbd_fk]; … … }
```

Figure 5.6: Adding *dbd_fk* schema to the existing query workload (*1, 2, 4*) that was detected as a duplicity to tested workload (*1, 4, 2*). In contrast to Figure 5.5, this figure also shows the list of schemata bounded to the workload.

QUERY WORKLOAD QUEUE

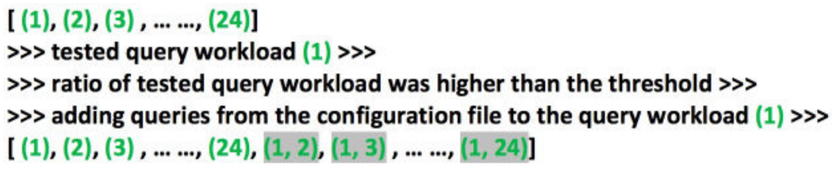
Since Python does not support a sorted map (key-value structure), an auxiliary queue of query workloads (FIFO – First In First Out) must be maintained to define the order of query workloads to test. At the initial phase of the tool run, the *query workload queue* is created as a sorted list of the query bucket's keys; this situation is shown in Figure 5.7. As the new query workloads are added to the query bucket, they are added into the query workload queue as well. To illustrate, this situation is shown in Figure 5.8.



```
{ (1): [dbd_basic, dbd_fk, dbd_customized]; (2): [dbd_basic, dbd_fk, dbd_customized];
(3): [dbd_basic, dbd_fk, dbd_customized]; …; (24): [dbd_basic, dbd_fk, dbd_customized] }
>>> creating query workload queue from keys of query bucket>>>
[ (1), (2), (3) , … …, (24)]
```

Figure 5.7: Initial phase of the *query workload queue* at the beginning of the query run.

4. Then the query workloads with two queries and likewise the others.



```
[ (1), (2), (3) , … …, (24)]
>>> tested query workload (1) >>>
>>> ratio of tested query workload was higher than the threshold >>>
>>> adding queries from the configuration file to the query workload (1) >>>
[ (1), (2), (3) , … …, (24), (1, 2), (1, 3) , … …, (1, 24)]
```

Figure 5.8: Adding items to the *query workload queue*.

5.3 Output

The DBPerfComp tool creates a new schema named *monitoring_output* in a database along with three special tables that are located in it.

The first table is named *monitoring_results* and contains data about a run of each query iteration. With a specific query label, the data is retrieved from the system tables *dc_requests_issued*, *dc_requests_completed*, *dc_resource_acquisitions*, *dc_resource_releases* and *execution_engine_profiles*. System tables descriptions are in Section 3.4. Actually, the tool retrieves the data from the snapshots of these system tables. The snapshots are used because of the Vertica retention policy which may cause disappearing of the required data from the system tables in a time of their retrieving. The query that retrieves the data is named *monitoring_snap.sql*. The content of the *monitoring_results* table is listed in Appendix C.1.

The second table is named according to the *output_table_name* attribute in the configuration file. This table is not the same for all tests, because each test may need a different table; the number of table columns depends on the number of tables specified in the configuration file. In this thesis, the output table is called *test_design*. Above all outputs, this table is the main output of the DBPerfComp tool because it contains previously retrieved data from the *monitoring_results* table per query workload and per schema, and much more data. The content of this table is listed in Appendix C.2.

Similarly to the previous table, the name of the third table also depends on the *output_table_name* attribute in the configuration file. In this case, the name has prefix *bucket_*, so the table used in this thesis is named *bucket_test_design*. This table contains queries order for each query workload in the test and boolean attribute that indicates whether the query workload has been already processed. The content of this table is listed in Appendix C.3.

During the run of the DBPerfComp tool, the Database Designer creates two files (projections script, deployment script) for each design. These files are stored in a directory that is set in the configuration file.

The tool is logging all its activity during the process of testing. The logs are not only sent to the standard output but also stored into a file that is created in the *Logs* directory. A name of the log file has pattern *dbperfcomp_year-month-day_hour-minute-second.log*, and each log has a pattern [*Mode – Part of the mode*] *Comment*. An example of logs is shown in Appendix C.4.

5.4 Documentation

The DBPerfComp tool has three requirements that must be fulfilled to work properly. First, the tool needs Python's version 2.7 or higher. Second, the tool uses the *pyodbc* module for connecting to the database and for that reason the module must be installed. The third requirement is to provide sufficient permissions to storage directories and objects in a database.

The tool may be run with two arguments. First, the configuration file is specified with the argument *-cf* or *--conf_file*. This file must be stored in the directory *ConfigFiles*, and the default file is set to *dbd.yaml*. In that case, the argument may be skipped. The configuration file must be in the *YAML* format, and its structure is shown in Figure 5.9. Second, the mode is specified with the argument *-m* and *--mode*, and the default value is *TestDesign*. All possible values of the mode are *Compare*, *CopySchema*, *CreateSchema*, *Deployment*, *Design*, and *TestDesign*.

The first part of the YAML file sets attributes that specify a database.

- *is_cluster* – 1 = Database situated on a cluster, 0 = NOT situated on a cluster (possible values: 1,0). Sets different safety in a process of creating tables.
- *is_8_or_higher_version* – 1 = Vertica is in the 8.0 or higher version, 0 = is NOT (possible values: 1,0). Different usage in the *Test design* mode.
- *pyodbc_db_connection_string* – Connection string to a database (needed for the pyodbc module).
- *timeout* – Maximum time in seconds that the queries in the *Run Queries* and runs of *Test design* modes. The queries are killed after this timeout.

The *Run Queries* mode serves for running input queries on input schemata with input iteration of the queries. The *monitoring_results* table is the output of this mode and is described in Section 5.3. In brief, main monitored attributes are *response_ms*, *memory_allocated_kb*, *memory_used_kb* and *cpu_time*. As a result, the output contains information about queries performance on input schemata. Before running this mode, all tested schemata must be deployed in a database. All attributes from the configuration file that are related to this mode are as follow:

- *testName* – Name of the test.
- *runName* – Name of the specific run – must be unique for each run of the tool.
- *queries* – Queries to be tested (separated with SPACES). The queries must have label "_LABEL_".
- *schemas* – Schemata to be tested (separated with SPACES).
- *iteration* – Number of iterations of each query on each schema.

The *Create Schema* mode serves for creating a new schema in a database with an input script for the schema definition. Furthermore, the tool needs a path to the data files and a path to scripts with statements for copying data from the files to tables. All attributes from the configuration file that are related to this mode are listed below:

- *name* – Name of the schema to be created.
- *data_path* – Path to the directory where the data files are stored.
- *schema_path* – Path to the schema definition.
- *copy_query_path* – Path to the script with COPY statements.

The *Copy Schema* mode serves for creating a copy of any already created schema. This feature is available only in a Vertica's version 8.0 and higher. Since it uses a function *COPY_TABLE*, the tool needs a list of used tables. All attributes from the configuration file that are related to this mode are listed below:

- *tables* – Tables to be copied from the input schema.
- *schema_input* – Schema that the data is copied from.

- *schema_output* – Schema that the data is copied to.
- *schema_path* – Path to the schema definition.

The *Deployment* mode serves for deploying a script that contains DDL statements with new projections definitions. In addition, it enables replacing the schema that is stated in the definition with the new one (e.g. replace *... FROM schema1.LINEITEM...* with *... FROM schema2.LINEITEM...*). The output deployment script of the Database Designer is an example of a required script. All attributes from the configuration file that are related to this mode are listed as follow:

- *query_deployment_path* – Path to the deployment script.
- *previous_schema_occurs* – 1 = Previous schema occurs in the script – *... FROM old_schema.LINEITEM...* , 0 = Does not occur – *... FROM actual_schema.LINEITEM...* or *... FROM LINEITEM...*; Possible values: 1,0.
- *actual_schema_name* – Name of the schema that the script is to be deployed to.
- *previous_schema_name* – Name of the previous schema (if occurs) – this name will be replaced with actual_schema_name.

The *Design* mode serves for running a design process of the Database Designer that is described in Section 4.3. Attributes in the configuration file correspond to arguments of the Database Designer's functions (see Vertica documentation – `https://my.vertica.com/docs/8.0.x/HTML/` – chapter *Database Designer Functions*) are listed below:

- *design_name* – Name of the design.
- *design_schema* – Name of the schema that the tables are deployed to.
- *tables* – Tables that the Database Designer works on.
- *query_path* – Path to the queries.
- *queries* – Queries that the projections are optimized for.
- *type* – COMPREHENSIVE (projections redesign) or INCREMENTAL (additional projections).

- *objective* – Focused on QUERY (high performance), LOAD (small footprint) or BALANCED.
- *deploy_path* – Path to the deployment script.
- *deployment* – 1 = design is deployed, 0 = design is only created and not deployed (possible values: 1,0).
- *ksafe* – K-safety of the deployment (insert number). If a database is not situated in a cluster, the value must be 0.

The *Test Design* mode is described in Section 5.2 and uses two attributes from the previous *Design* mode – *query_path* and *deploy_path*. All other attributes from the configuration file that are related to this mode are listed below:

- *testName* – Name of the test.
- *max_set_depth* – Maximal number of queries in query workloads.
- *base_schema* – Base schema that is to be compared to.
- *base_schema_path* – Path to the base schema definition.
- *schemas* – Schemata that are compared with base_schema (separated with SPACES).
- *schemas_path* – Path to the schemata definitions (separated with SPACES).
- *queries* – Queries to be tested (separated with SPACES). The queries must have label "_LABEL_".
- *tables* – Tables to work with.
- *threshold* – Threshold of a ratio for adding new query workloads.
- *iteration* – Number of queries iterations in each test.
- *output_table_name* – Name of an output table in the monitoring_output schema that monitored data is stored to.
- *size_of_data_in_gb* – Size of schemata in GB.

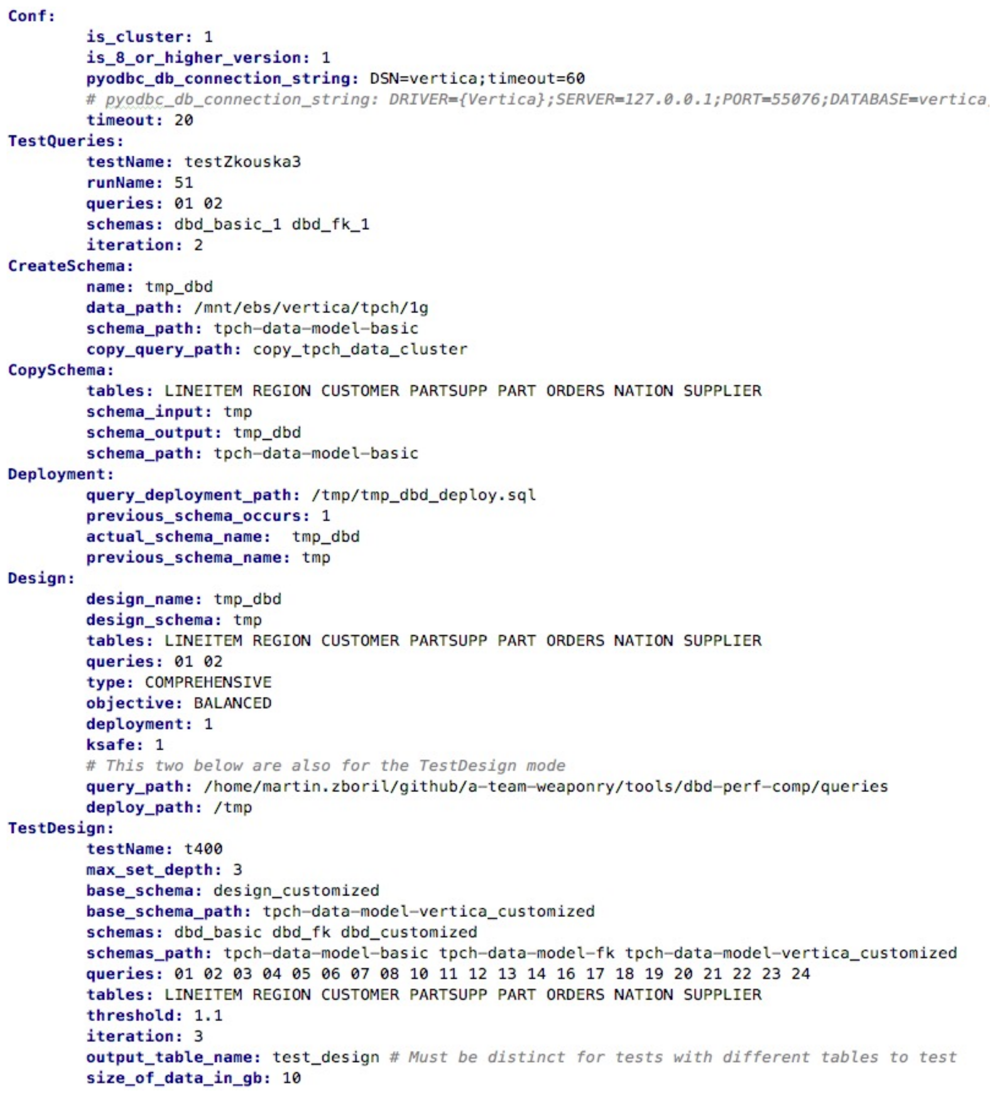
The *testname* and *runname* attributes, that occur in many modes, are important for results identification in the output database tables. For this reason, the *runname* must be unique for each run of the tool. The queries attributes may start with or without '/'. In the case of '/', the tool works with a direct path to the queries. In a case of no '/' at the beginning, the tool works with queries located in the directory *Queries*, not in the current directory of the tool. The directory layout is described in Section 5.5.

The DBPerfComp tool requires a configured connection to a Vertica database. Except for a connection string in the configuration file of the DBPerfComp tool, two files need to be specified. These files are described in the Vertica documentation[8] in chapter *Installing ODBC Drivers on Linux, Solaris, AIX, and HP-UX*:

> *"You must configure the ODBC driver before you can use it. There are two required configuration files:*
>
> - *The odbc.ini configuration file defines the Data Source Names (DSNs) that tell the ODBC how to access your Vertica databases. See Creating an ODBC Data Source Name for instructions to create this file.*
> - *The vertica.ini configuration file defines some Vertica-specific settings required by the drivers. See Required ODBC Driver Configuration Settings for Linux and UNIX for instructions to create this file."*

The user, who is specified in the connection, must have sufficient permissions for objects in a database that the DBPerfComp tool works with and for triggering the Database Designer. Moreover, they must have sufficient permissions for directories and files in the file system that the DBPerfComp tool works with. Examples of required permissions are read permission for queries to test, read permission to data files, write permission for a directory where deployment scripts of the Database Designer are stored.

The exemplary configuration file is shown in Figure 5.9.



```
Conf:
        is_cluster: 1
        is_8_or_higher_version: 1
        pyodbc_db_connection_string: DSN=vertica;timeout=60
        # pyodbc_db_connection_string: DRIVER={Vertica};SERVER=127.0.0.1;PORT=55076;DATABASE=vertica
        timeout: 20
TestQueries:
        testName: testZkouska3
        runName: 51
        queries: 01 02
        schemas: dbd_basic_1 dbd_fk_1
        iteration: 2
CreateSchema:
        name: tmp_dbd
        data_path: /mnt/ebs/vertica/tpch/1g
        schema_path: tpch-data-model-basic
        copy_query_path: copy_tpch_data_cluster
CopySchema:
        tables: LINEITEM REGION CUSTOMER PARTSUPP PART ORDERS NATION SUPPLIER
        schema_input: tmp
        schema_output: tmp_dbd
        schema_path: tpch-data-model-basic
Deployment:
        query_deployment_path: /tmp/tmp_dbd_deploy.sql
        previous_schema_occurs: 1
        actual_schema_name:  tmp_dbd
        previous_schema_name: tmp
Design:
        design_name: tmp_dbd
        design_schema: tmp
        tables: LINEITEM REGION CUSTOMER PARTSUPP PART ORDERS NATION SUPPLIER
        queries: 01 02
        type: COMPREHENSIVE
        objective: BALANCED
        deployment: 1
        ksafe: 1
        # This two below are also for the TestDesign mode
        query_path: /home/martin.zboril/github/a-team-weaponry/tools/dbd-perf-comp/queries
        deploy_path: /tmp
TestDesign:
        testName: t400
        max_set_depth: 3
        base_schema: design_customized
        base_schema_path: tpch-data-model-vertica_customized
        schemas: dbd_basic dbd_fk dbd_customized
        schemas_path: tpch-data-model-basic tpch-data-model-fk tpch-data-model-vertica_customized
        queries: 01 02 03 04 05 06 07 08 10 11 12 13 14 16 17 18 19 20 21 22 23 24
        tables: LINEITEM REGION CUSTOMER PARTSUPP PART ORDERS NATION SUPPLIER
        threshold: 1.1
        iteration: 3
        output_table_name: test_design # Must be distinct for tests with different tables to test
        size_of_data_in_gb: 10
```

Figure 5.9: Structure of YAML file with exemplary values.

5.5 Directory layout

The directory layout of the tool is shown in Figure 5.10. The directories *ConfigFiles* and *Queries* need to be created and fulfilled with appropriate files before a tool run. Directories *ExplainFiles* and *Logs* are dynamically created during a tool run. Directory *Requirement* contains *pyodbc* python module that must be installed before a tool run and two exemplary configuration files that set database connection parameters. Next, the tool layout consists of python files that were implemented as a part of this thesis. The main file is named *db_comp_perf.py*, and the additional files are *designer.py* and *parser_config.py*.

```
├── ConfigFiles
│   └── dbd.yaml
├── ExplainFiles
├── Logs
├── Queries
│   ├── 01.sql
│   ├── 02.sql
│   ├── bucket_create.sql
│   ├── bucket_insert.sql
│   ├── copy_tpch_data.sql
│   ├── copy_tpch_data_cluster.sql
│   ├── monitor_create.sql
│   ├── monitor_design_create.sql
│   ├── monitor_projections_size.sql
│   ├── monitor_snap.sql
│   ├── snap_analyze.sql
│   ├── snap_create.sql
│   ├── snap_insert.sql
│   ├── snap_truncate.sql
│   ├── tpch-data-model-basic.sql
│   ├── tpch-data-model-fk.sql
│   └── tpch-data-model-vertica_customized.sql
├── Requirement
│   ├── odbc.ini
│   ├── pyodbc-2.1.9.zip
│   └── vertica.ini
├── README.md
├── db-comp-perf.py
├── designer.py
└── parser_config.py
```

Figure 5.10: Structure of the DBPerfComp tool.

6 Analysis

6.1 TPC-H model

This section introduces TPC-H model that is used in Chapter 6 in analyses of Database Designer influence on database objects and in Chapter 7 as an input for the DBPerfComp tool. The model is described in the official TPC-H specification created by the TPC (Transaction Processing Performance Council) organization as:

> *"The TPC Benchmark™H (TPC-H) is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaining a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that:*
>
> - *Examine large volumes of data;*
> - *Execute queries with a high degree of complexity;*
> - *Give answers to critical business questions.*
>
> *(...)*
> *TPC Benchmark™ H is comprised of a set of business queries designed to exercise system functionalities in a manner representative of complex business analysis applications. These queries have been given a realistic context, portraying the activity of a wholesale supplier to help the reader relate intuitively to the components of the benchmark."*[24]

TPC-H schema contains eight tables that are shown in Figure 6.1. All columns of each table have the same prefix that is written in parentheses after table names, e.g. *PART (P_)*. Furthermore, the schema in Figure 6.1 contains information about the number of table rows in numbers below tables names. Six tables have a variable number of rows that depends on a scale factor (SF) that determines the size of the schema in gigabytes.

TPC-H model includes 22 queries that are described in the official TPC-H specification [24]. For each query, the specification contains its definition, description, business question, substitution parameters, query validation and sample output.

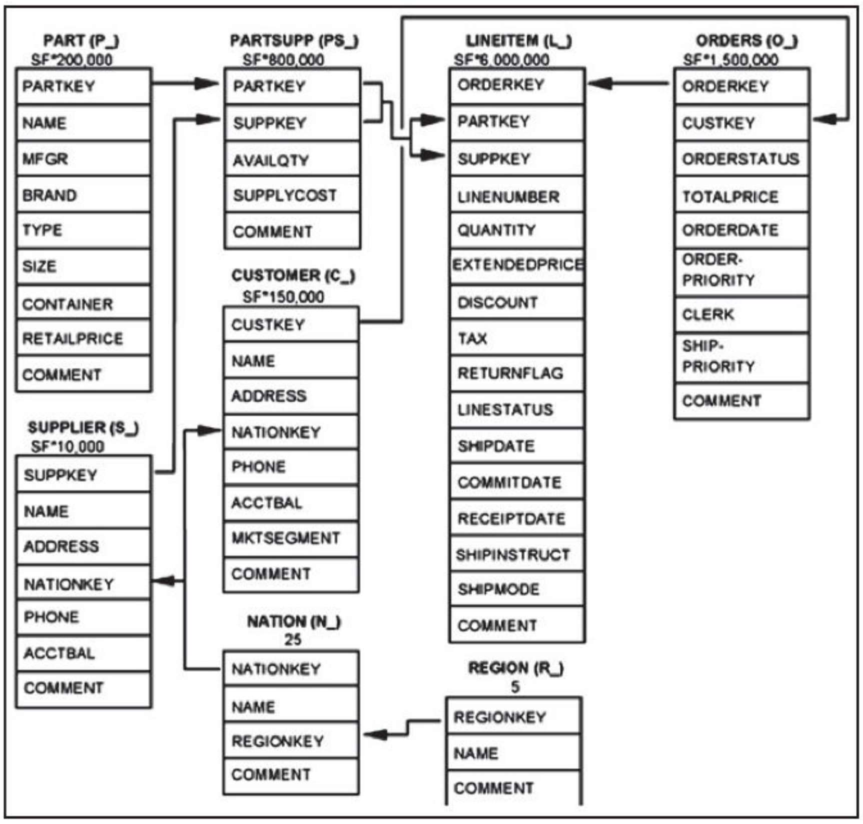


Figure 6.1: TPC-H schema.[24]

6.2 Hardware and software

All measurements in Chapter 6 were performed with the same software and hardware. Database was run on Vertica 8.0.1 using Red Hat Enterprise Linux 7.3 operating system. The database was situated on a four-nodes cluster where the nodes were virtualized machines with following parameters:

- CPU: 2 cores, 2 threads (HyperThreading off)
- RAM: 10 GB
- HDD: 200 GB, LVM
- Virtualization solution: OpenStack

The physical machine where the nodes were situated had following parameters:

- Dell R720xd
- CPU: 2x E5-2697v2@2.7GHz, 24 cores, 48 threads (HyperThreading)
- RAM: 384 GB DDR3@1333MHz
- HDD: 24x 600 GB 10k 2.5" 6 Gbitps
- RAID Controller: PERC H710p
- NetCard: 2x Intel X520 DA 10 GbE Dual Port
- Disk layout: 64k strip size, RAID6 23x 600 GB + 1x 600 GB HotSpare, No read ahead, Write Back cache, Full Initialization
- HW encryption off

6.3 Database Designer's influence on database objects

Database Designer uses different database objects during designing process. For that reason, the tool may influence their availability, since it may lock them for an undefined time. This implies that Database Designer may influence the performance of any database. The motivation for analysis of this influence is to investigate whether companies that use Vertica may run Database Designer during business hours, since database performance and availability may be affected. There are also companies on the market that provide Vertica to their customers and may set whether the customers are allowed running Database Designer or not.

Each analysis used all tables and queries from TPC-H model. Tables were 10 GB large; therefore, the scale factor of TPC-H model was 10. In a case of each Database Designer usage, the designing process did not include deployment of the design. For a time of every designing process, queries from TPC-H model were simultaneously running in a background on the tables that participated in the design. These running queries simulated stress on database. Three scenarios in database were used in the analyses:

A. *OutTPCH* – TPC-H tables were situated in a **different schema** than the majority of objects in database catalog (small tables containing one row). + **TPC-H queries** were running in the background on TPC-H tables.

B. *InTPCH* – TPC-H tables were situated in the **same schema** as the majority of objects in database catalog. + **TPC-H queries** were running in the background on TPC-H tables.

C. *Insert* – TPC-H tables were situated in a **different schema** than the majority of objects in database catalog. + The small tables were in the process of creation; in other words, there were running **CREATE TABLE** and **INSERT** statements in the background.

The analyses also explore the influence of different numbers of objects in database catalog. For increasing the number of objects, small tables with one row were created. The analyses work with four different number of objects:

1. approximately 250,000 objects in database catalog,
2. approximately 500,000 objects in database catalog,
3. approximately 750,000 objects in database catalog,
4. approximately 1,000,000 objects in database catalog.

This chapter analyzes the influence of different scenarios in database in Section 6.3.1 and influence of Database Designer in several ways that are separated into Sections 6.3.2 – 6.3.4. Results are gathered into the Excel file that is attached to the thesis.

6.3.1 Tables location

The first analysis explores the influence of tables location on database performance and compares durations of queries runs in the three previously described scenarios in database (*OutTPCH*, *InTPCH*, and *Insert*). Measurements were performed once with 250,000 objects in database and once with 1,000,000 objects.

Results of measurements are shown in Figures 6.2 and 6.3. Apparently, the different scenarios only slightly influence performed queries.

With 250,000 objects, the smallest times of average query durations and average catalog responses are in the *OutTPCH* scenario, then in the *Insert* and the highest times are in the *InTPCH*. With 1,000,000 objects, values increased, but the ordering stayed the same.

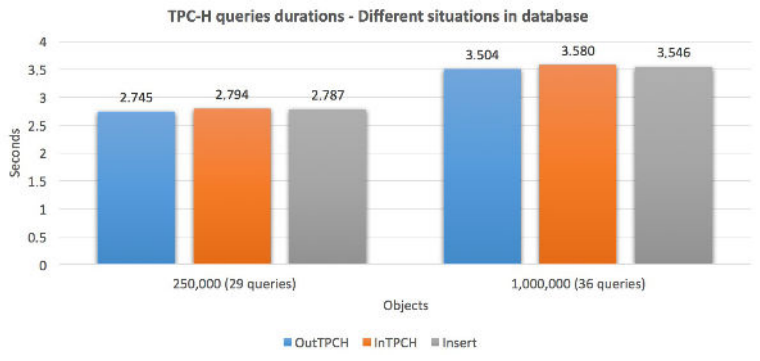


Figure 6.2: Comparison of queries durations within different scenarios.

According to Figure 6.3, catalog response time depends on the number of objects in the catalog; the catalog response time is four times higher at 1,000,000 objects than at 250,000 objects. For that reason, it is assumed that the catalog is gone through sequentially.

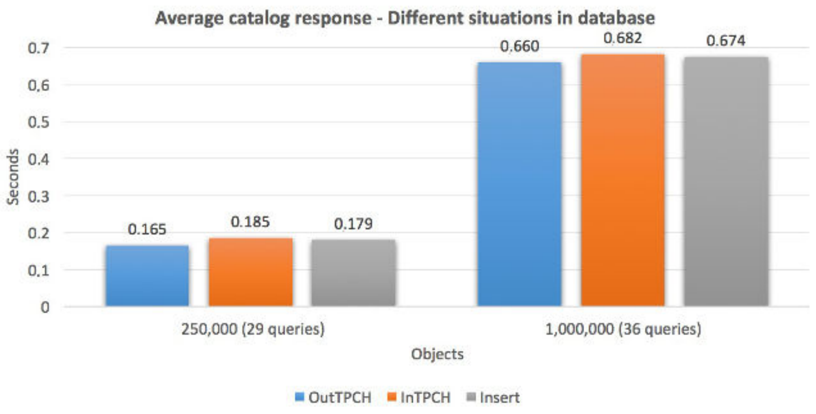


Figure 6.3: Comparison of catalog responses within different scenarios.

6.3.2 TPC-H queries, Database Designer and different catalog size

This analysis explores the difference between running queries on tables that simultaneously participate in designing process of Database Designer and on tables that do not participate in designing process. Besides, the analysis explores how numbers of objects in database catalog influence running queries. Measurements are performed in the *OutTPCH* scenario.

At each level of amounts of objects, measurements with designing process were performed first. Then, the number of successfully run queries was retrieved from system tables. Subsequently, the same number of queries was performed without a simultaneous run of the designing process. An output of monitoring includes average duration, average catalog response, and total time of each measurement.

Results of measurements are shown in Figures 6.4 and 6.5 where amounts of performed queries are written in parentheses on a horizontal axis. Average durations of queries are shown in Figure 6.4. Clearly, increasing numbers of objects relate to the duration of queries. Equally, an occurrence of simultaneous designing process increase duration of performed queries. Catalog response times are shown in Figure 6.5. Comparing to the previous figure, the difference in catalog response times between performing queries with a simultaneous designing process and without the process is smaller than the difference in duration of queries. But similarly, catalog response times grow with increasing numbers of objects. Table with measured data is stored in the attached Excel file.

6.3.3 Steps of designing process

Locks that occur during designing process, which is described in Section 4.3, are analysed here. Locks created during separate steps were monitored after each call of Database Designer functions (see Section 4.3.1). The monitoring was performed with usage of monitoring query included in Appendix D and system tables *dc_lock_attempts*, *dc_lock_releases* and *query_requests* described in Section 3.4. To identify all locks that are created during the designing process, the query uses as identifiers a specific user who calls the functions and a specific design name that is defined in the first step of designing process.

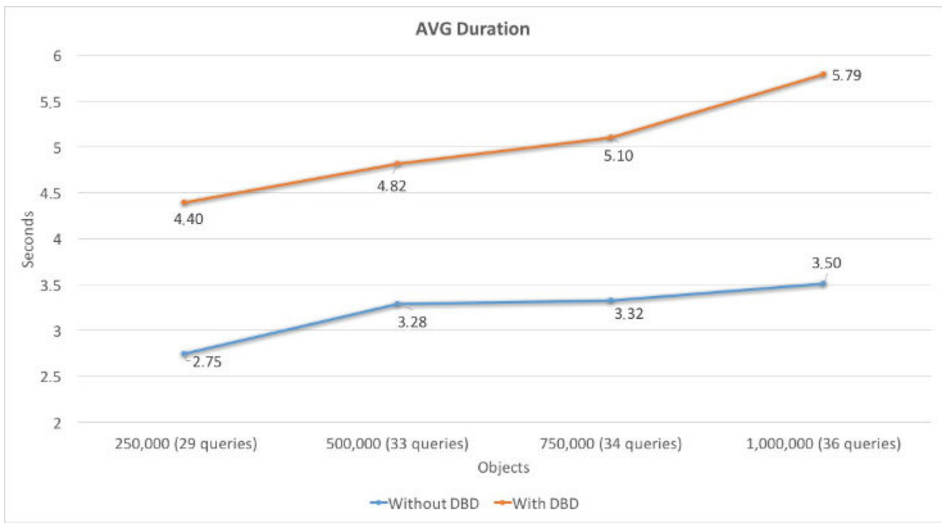


Figure 6.4: Comparison of queries average durations with the simultaneous running of designing process (Database Designer) and without designing process.

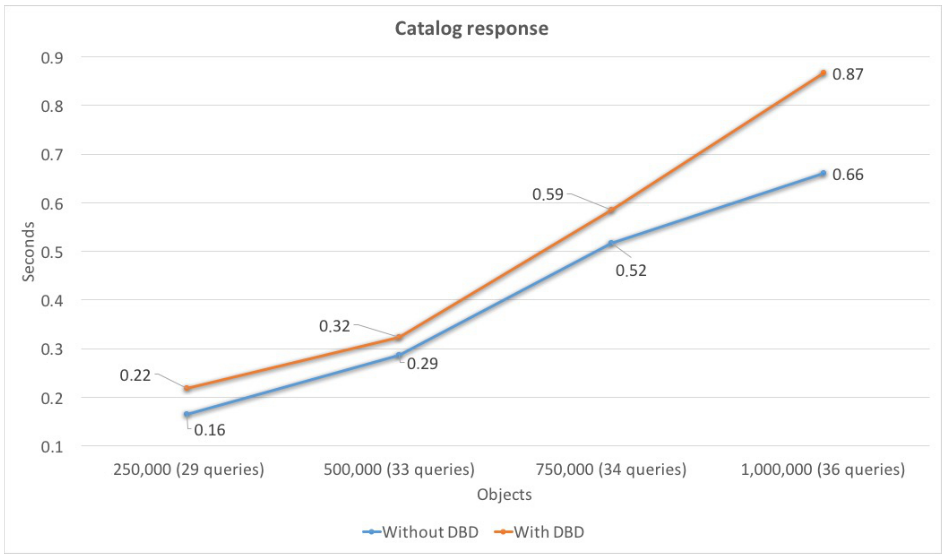


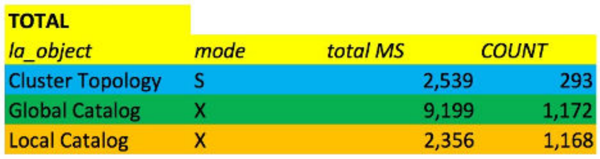
Figure 6.5: Comparison of queries catalog response times with simultaneously run of designing process (Database Designer) and without designing process.

An output of the query is listed as follows:

- object name,
- mode,
- total lock time in ms,
- total count of locks,
- start time of the first lock,
- end time of the last lock.

In this analysis, a number of objects in database catalog was 1,000,000, and the *OutTPCH* scenario was used. In general, a number of objects and used scenario were not significant, since database objects with locks were only found and particular steps of the designing process were compared.

Results of this analysis are situated in the attached Excel file. For each function of designing process, all influenced objects are listed in the sheet together with measured values of the monitoring query. Three influenced objects occurred in all steps of the designing process: *Cluster Topology*, *Global Catalog*, and *Local Catalog*[1]. Besides, many locks occurred on tables that were created during designing process of Database Designer; because these tables were dropped along with dropping the design, they were not significant for database availability. The three objects, which occurred in all steps, are shown together with their lock modes (see Appendix A) and locks in milliseconds in Figure 6.6.



TOTAL			
la_object	*mode*	*total MS*	*COUNT*
Cluster Topology	S	2,539	293
Global Catalog	X	9,199	1,172
Local Catalog	X	2,356	1,168

Figure 6.6: Objects with locks that occur in each step of designing process.

The highest number of locks had *Global Catalog*. In this measurement, the delay due to locks was not significant, since the total delay was approximately 9 seconds at *Global Catalog*, but the total run of Database Designer functions lasted 326 seconds. It means that the locks lasted only 2.7% of the run time. Moreover, total number of locks was 1,172, and in that case, the average delay of one locks was 8 milliseconds. This implies that the database user does not even notice any lock during their work.

1. Locks on *Cluster Topology*, *Global Catalog*, and *Local Catalog* also occurred during basic CRUD operations. Detail results are in Appendix E.

Distribution of locks between particular steps is shown in Figure 6.7. The majority of locks belong to the designing step (function *DESIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY*). Design dropping is the second step with the highest number of locks, since it must drop data that were created during designing process. Design creation is the third step with the highest number of locks, because it creates many initial records in database. In these measurements, eight tables and twenty-two queries were added into the design. The locks that were created during the addition of tables and queries were not significant, since they were incomparably smaller than the locks in the previously mentioned steps. In comparison with the designing step, 494 tables or 1,469 queries need to be added to have the same amount of locks on the object *Local Catalog*[2].

6.3.4 Different catalog size

The last analysis explores the influence of different numbers of objects and different scenarios in database (*OutTPCH*, *InTPCH*, and *Insert*) on number of locks created on database objects during designing process. This analysis uses findings from the previous analysis; therefore, it compares only *Cluster Topology*, *Global Catalog*, and *Local Catalog* objects.

The analysis includes 12 individual measurements. Each measurement has its own sheet in the attached Excel file, and each sheet contains the analysis of particular steps during designing process. The sheets are defined with catalog size identifiers (*250*, *500*, *750*, *1000*) and with *OutTPCH*, *InTPCH*, and *Insert* for a different scenario in database. The overview of all results is shown in Figure 6.12. Results are separated into four graphs in Figures 6.8 – 6.11.

Figure 6.8 shows all 12 measurements and their locks in milliseconds on the object *Cluster Topology*. In measurements *OutTPCH*, numbers of locks were approximately constant for all numbers of objects. In measurements *InTPCH*, numbers of locks increased a little but also stayed approximately constant for different numbers of objects. These results show that database availability is slightly influenced by the size of schema where the tables are located; the smaller size of schema with queried tables is, the higher database availability is provided. Unlike the previous scenarios, a growth of locks occurred in measurements *Insert* with an increasing number of object. For 250,000 and 500,000 objects, numbers of locks were approximately the same. For 750,000 objects, a number of locks increased a little.

2. The object with the smallest number of locks during the designing step.

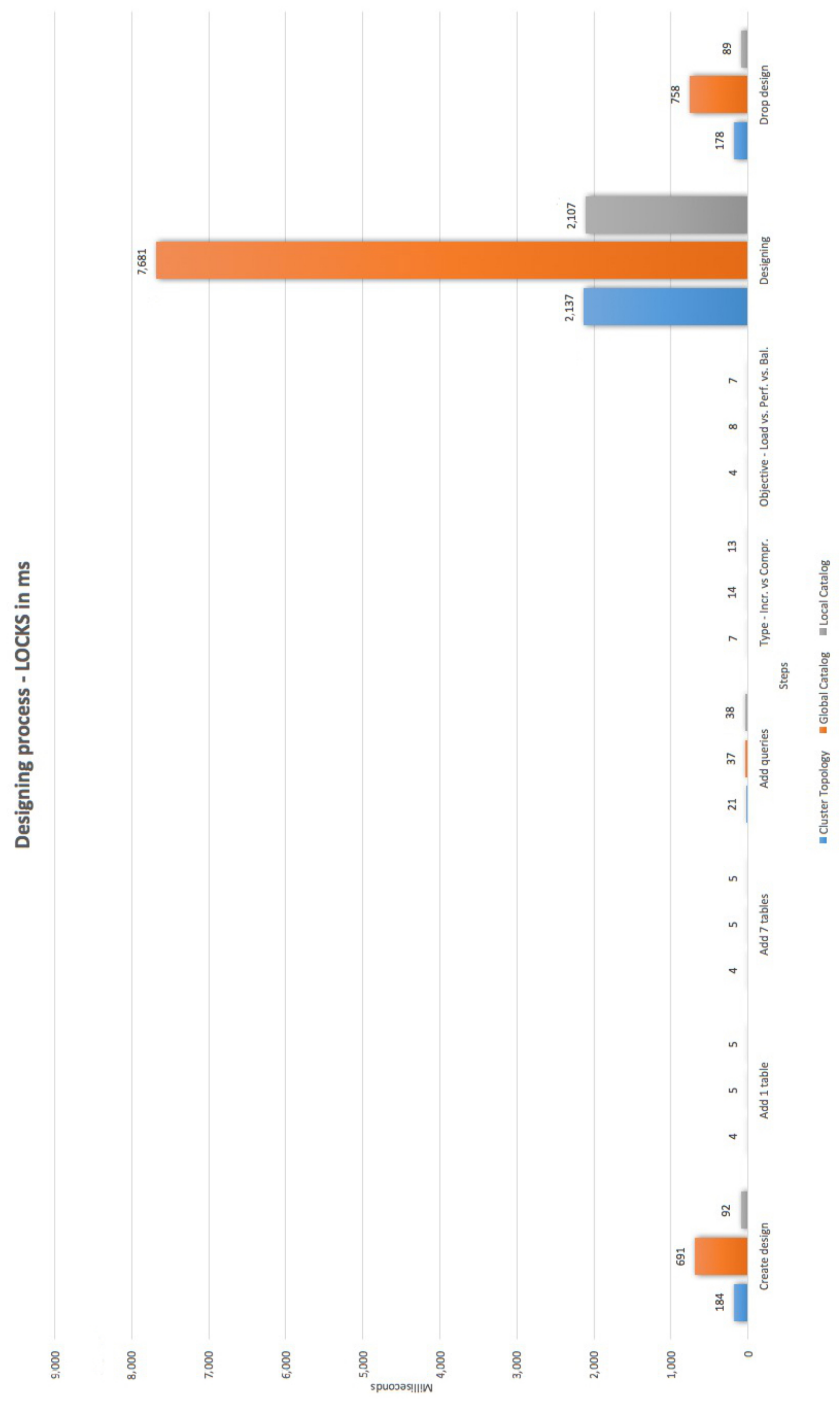


Figure 6.7: Distribution of locks between particular steps of designing process.

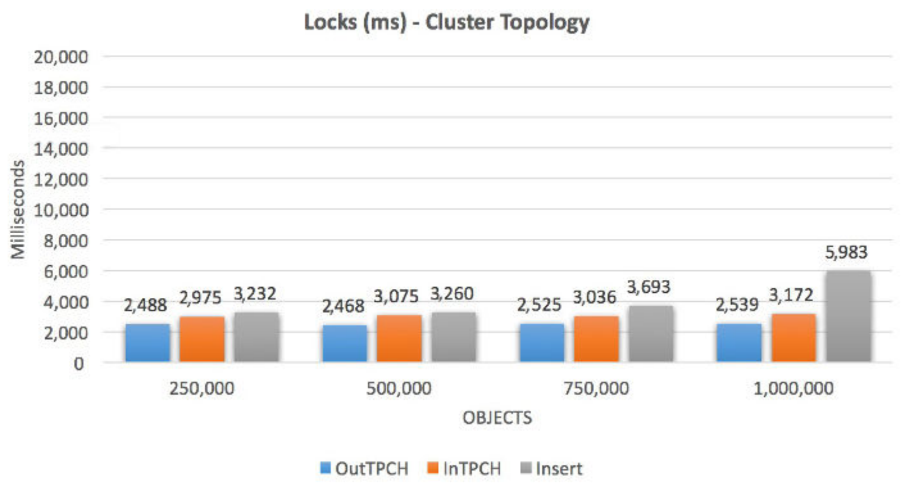


Figure 6.8: Locks on the *Cluster topology* object during the designing process.

Then for 1,000,000 objects, a boom in a number of locks occurred as there were almost twice more locks. The boom indicates that the number of objects in database influences database availability during the Database Designer run. Furthermore, it shows that *CREATE TABLE* and *INSERT* statements influence the availability much more than only running *SELECT* statements; the reason is that the data must be distributed across all node of the cluster. For that reason, the results might be different if database is run situated on single machine, not cluster. Overall, different scenarios are significant for numbers of locks on the *Cluster Topology* object.

Relations between measurements on the *Global Catalog* were the same as on the *Cluster Topology*; the only difference was that values of all locks were approximately three times higher, as it is seen in Figure 6.9. Since the results are similar, their interpretation is skipped here.

Results of measurements on the object *Local Catalog* are shown in Figure 6.10. Apparently, the results are approximately the same for all measurements. So that locks on the object *Local Catalog* are not influenced by numbers of the objects and with *OutTPCH*, *InTPCH* and *Insert* scenarios.

All measurements for the *Insert* scenario are shown in Figure 6.11. This scenario was chosen for the comparison of the three objects, since it has the highest number of locks. Overall, an influence of the cluster is recognizable in the figure, because both the *Global Catalog* and the *Cluster Topology* had an increasing trend in numbers of locks. The *Local Catalog* holds meta-data of only one node, so the cluster has minimal influence there.

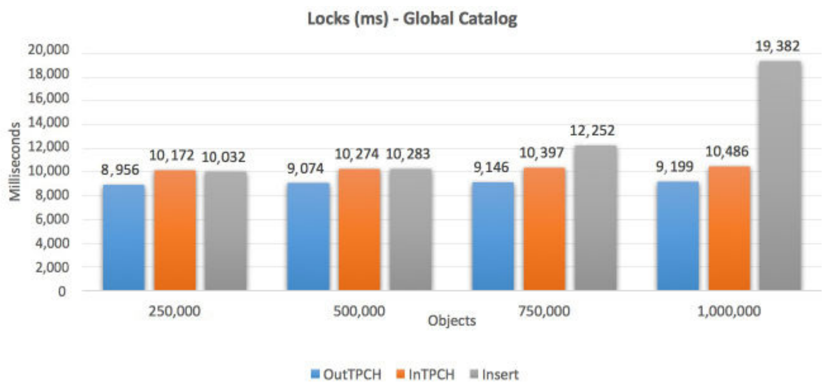


Figure 6.9: Locks on the *Global catalog* object during the designing process.

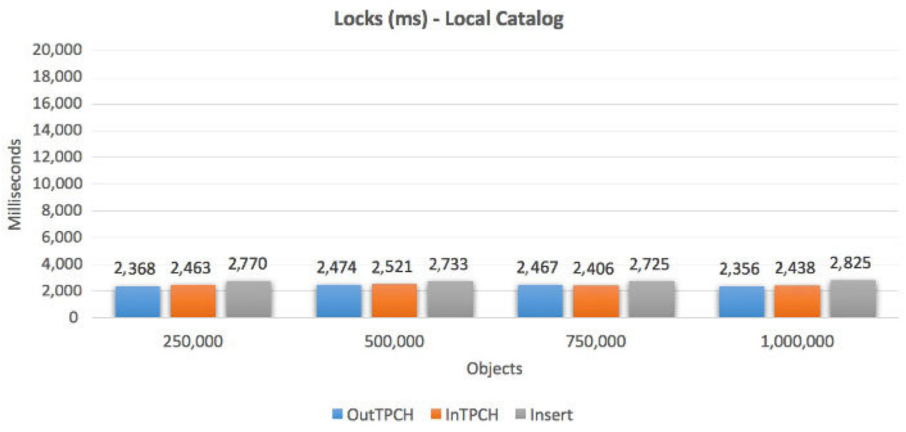


Figure 6.10: Locks on the *Local catalog* object during the designing process.

Overview of all results in Figure 6.12 provides information about the comparison between locks duration (in milliseconds in the overview) and total designing process duration (in seconds). The locks from the measurement with 1,000,000 objects, in the *Insert* scenario and on the object *Global Catalog* are used for the comparison, because it has the highest amount of locks. In this case, the designing process lasted 445 seconds, but the locks existed only for 19.4 seconds, i.e. 4.3% of the designing process. In this measurement, average lock lasted 16.2 milliseconds, and in all measurements, the highest average lock lasted 20 milliseconds.

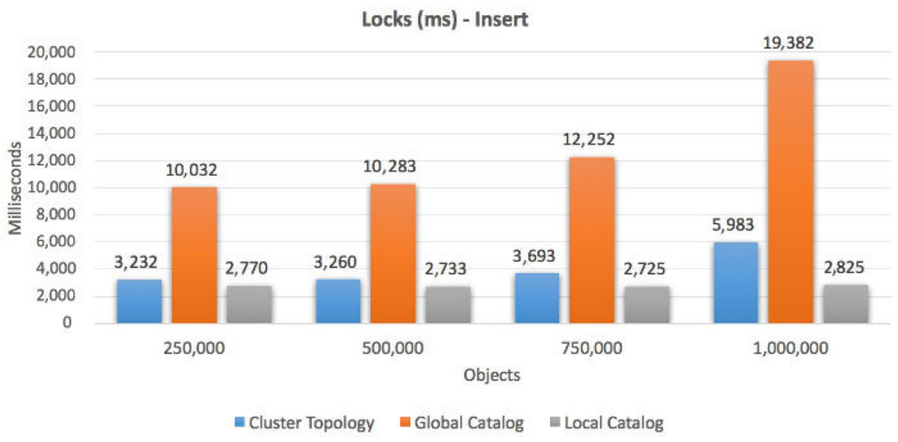
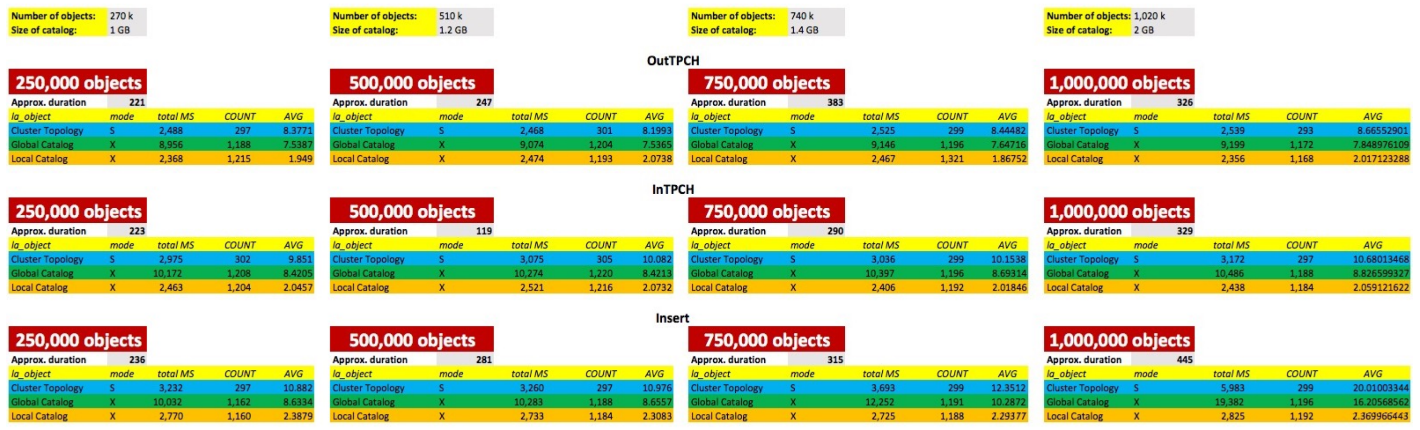


Figure 6.11: Measurement for the *Insert* scenario.

From the previous findings, it may be concluded that users of Vertica database are not influenced by the designing process, because the locks are not continual and duration of particular locks are too small to affect database availability negatively. Particular locks would have to last seconds to affect the availability.



Number of objects:	270 k
Size of catalog:	1 GB

Number of objects:	510 k
Size of catalog:	1.2 GB

Number of objects:	740 k
Size of catalog:	1.4 GB

Number of objects:	1,020 k
Size of catalog:	2 GB

OutTPCH

250,000 objects

Approx. duration 221

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	2,488	297	8.3771
Global Catalog	X	8,956	1,188	7.5387
Local Catalog	X	2,368	1,215	1.949

500,000 objects

Approx. duration 247

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	2,468	301	8.1993
Global Catalog	X	9,074	1,204	7.5365
Local Catalog	X	2,474	1,193	2.0738

750,000 objects

Approx. duration 383

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	2,525	299	8.44482
Global Catalog	X	9,146	1,196	7.64716
Local Catalog	X	2,467	1,321	1.86752

1,000,000 objects

Approx. duration 326

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	2,539	293	8.66552901
Global Catalog	X	9,199	1,172	7.848976109
Local Catalog	X	2,356	1,168	2.017123288

InTPCH

250,000 objects

Approx. duration 223

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	2,975	302	9.851
Global Catalog	X	10,172	1,208	8.4205
Local Catalog	X	2,463	1,204	2.0457

500,000 objects

Approx. duration 119

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,075	305	10.082
Global Catalog	X	10,274	1,220	8.4213
Local Catalog	X	2,521	1,216	2.0732

750,000 objects

Approx. duration 290

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,036	299	10.1538
Global Catalog	X	10,397	1,196	8.69314
Local Catalog	X	2,406	1,192	2.01846

1,000,000 objects

Approx. duration 329

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,172	297	10.68013468
Global Catalog	X	10,486	1,188	8.826599327
Local Catalog	X	2,438	1,184	2.059121622

Insert

250,000 objects

Approx. duration 236

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,232	297	10.882
Global Catalog	X	10,032	1,162	8.6334
Local Catalog	X	2,770	1,160	2.3879

500,000 objects

Approx. duration 281

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,260	297	10.976
Global Catalog	X	10,283	1,188	8.6557
Local Catalog	X	2,733	1,184	2.3083

750,000 objects

Approx. duration 315

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	3,693	299	12.3512
Global Catalog	X	12,252	1,191	10.2872
Local Catalog	X	2,725	1,188	2.29377

1,000,000 objects

Approx. duration 445

la_object	mode	total MS	COUNT	AVG
Cluster Topology	S	5,983	299	20.01003344
Global Catalog	X	19,382	1,196	16.20568562
Local Catalog	X	2,825	1,192	2.369966443

Figure 6.12: Overview of results from particular measurements where number of objects in database catalog differs, as well as a scenario in a database.

7 Data evaluation

The test presented in Chapter 5 targets weaknesses of Database Designer in the form of designed projections that are less optimal for running given queries than projections that were not designed by Database Designer.

The experiment was performed twice to compare the behavior of Vertica on two clusters that differ in performance. Results from the both clusters are described in Sections 7.2.1 and 7.2.2. The first cluster's (identifier: *cluster1*) parameters were described in Section 6.2. The second cluster (identifier: *cluster2*) is of parameters:

- CPU: 10 cores, 10 threads (HyperThreading off)
- RAM: 88.5 GB
- HDD: 2800GB, LVM
- Virtualization solution: OpenStack (the physical machine was the same as at *cluster1*)

On the *cluster2*, the database resource pool where the experiment run was limited to 10 GB in order to simulate the *cluster1's* RAM. Equally as in Chapter 6, measurements in this chapter use 8.0.1 version of Vertica and Red Hat Enterprise Linux 7.3 version of the operating system.

The whole evaluation is divided into several parts. Measured data from output tables of DBPerfTool are evaluated in Section 7.2. Based on this data, one query is identified as a bottleneck for Database Designer and is described in Section 7.3. Next, Section 7.4 includes the analysis of projections optimized by Database Designer for this query. Investigation of a query plan of the bottleneck query is performed in Section 7.5, and investigation of operations executed within a run of this query is performed in Section 7.6. Manual optimizations of designed projections for the bottleneck query are proposed and compared in Section 7.7.

7.1 Test setting

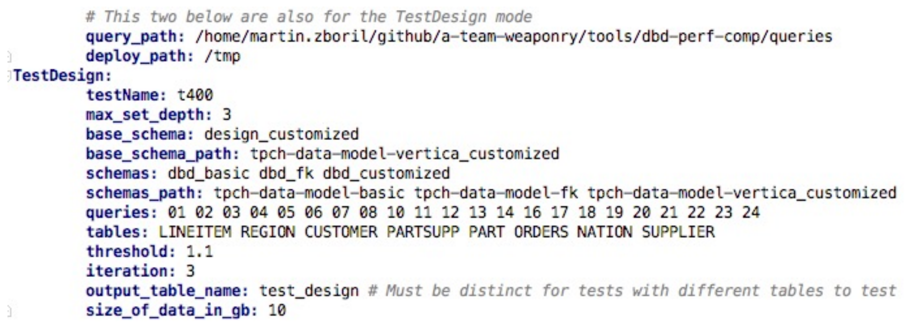
TPC-H model (tables, queries) described in Section 6.1 was used for the test. The total size of queried tables in each schema was 10 GB[1]. Queries

1. The standard size in production servers in GoodData s.r.o.

9 and 15 of TPC-H model were not used because a problem[2] occurred in retrieving data of these two queries from system tables. Instead, Queries *23* and *24* that use TPC-H tables were created for this experiment and are shown in Appendix F. The principle of both queries is to create complex queries that contain many *join* operations (including two largest tables *Lineitem* and *Orders*), many predicates and many columns in *GROUP BY* clauses.

The timeout was set to 30 seconds[3]. The configuration file with attributes that were used in this test is given in Figure 7.1. Selected attributes used in the configuration file are:

- Threshold = 1.1 – Query workloads whose response time is worse by 10 percent on a given schema than on the base schema.
- Max depth set = 3 – The maximum query workload depth was set to 3, i.e. there might be three-tuples of queries at maximum.
- Iteration = 3 – Each query was run three times.



```
        # This two below are also for the TestDesign mode
        query_path: /home/martin.zboril/github/a-team-weaponry/tools/dbd-perf-comp/queries
        deploy_path: /tmp
TestDesign:
        testName: t400
        max_set_depth: 3
        base_schema: design_customized
        base_schema_path: tpch-data-model-vertica_customized
        schemas: dbd_basic dbd_fk dbd_customized
        schemas_path: tpch-data-model-basic tpch-data-model-fk tpch-data-model-vertica_customized
        queries: 01 02 03 04 05 06 07 08 10 11 12 13 14 16 17 18 19 20 21 22 23 24
        tables: LINEITEM REGION CUSTOMER PARTSUPP PART ORDERS NATION SUPPLIER
        threshold: 1.1
        iteration: 3
        output_table_name: test_design # Must be distinct for tests with different tables to test
        size_of_data_in_gb: 10
```

Figure 7.1: Test setting.

Schemata that were used in the test:

- **Design_customized**
  - ***Base schema***
  - Information in this schema: Primary keys, Foreign keys, Segmentation.

2. Since the problem with Queries *9* and *15* was not caused by Database Designer, but occurred during retrieving data from system tables, the problem was not further investigated.
3. This specific time was set based on previous findings that were retrieved during manual runs of tested queries.

  - This schema is **NOT** used for designing process of Database Designer.
  - Superprojections are created automatically with a creation of tables.

- **DBD_customized**
  - Information in this schema: Primary keys, Foreign keys, Segmentation.
  - This schema is used for designing process of Database Designer.
  - Test is run on designed projections, not on the template schema.

- **DBD_fk**
  - Information in this schema: Primary keys, Foreign keys.
  - This schema is used for designing process of Database Designer.
  - Test is run on designed projections, not on the template schema.

- **DBD_basic**
  - Information in this schema: None.
  - This schema is used for designing process of Database Designer.
  - Test is run on designed projections, not on the template schema.

7.2 Results

7.2.1 Cluster 1

The first test was performed on *cluster1*, and its results are shown in Table 7.1. Rows represent different combinations of query workloads and tested schemata (hereinafter referred to as the *"query-schema set"*). The first row contains all tested *query-schema sets*. The second row involves combinations of all query workloads and *Design_customized* schema, i.e. a number of particular query workloads that were created during the tool run. The remaining rows contain query workloads for tested schemata *DBD_basic*, *DBD_fk*, and *DBD_customized*. The first column presents the total number of *query-schema sets*, and the second column presents the number of *query-schema sets* whose schemata designed by Database Designer were less optimal than *Design_customized* schema (hereinafter referred to as the *"non-optimal query-schema sets"*); i.e. response time of a tested query workload was higher on a tested schema designed by Database Designer than on *Design_customized*. Moreover, it means, that the ratios of a tested *query-schema sets* were higher than the threshold. Complete results are in the Excel file that is attached to the thesis. The findings retrieved from Table 7.1 are:

- For each query workload, *Design_customized* schema was created and tested. For that reason, only 1,118 *query-schema sets* belonged to tested schemata designed by Database Designer (1,861 *query-schema sets* minus 743 query workloads at *Design_customized* schema);
- 393 *query-schema sets* out of 1,118 (schemata designed by Database Designer) were non-optimal. It means, Database Designer created non-optimal designs (projections) in 33% cases.
- 1,118 *query-schema sets* consist of: 58.3% records (652) that belonged to *DBD_basic* schema, 28.2% (315) to *DBD_fk* and 13.5% (151) to *DBD_customized*;
- 393 non-optimal *query-schema sets* consist of: 71.8% records (282) that belonged to *DBD_basic*, 19% (75) to *DBD_fk* and 9.2% (36) to *DBD_customized*.

	Total	**Ratio > Threshold**
Combinations: Query workload × Schema (with *Design_customized*)	1,861	**393** (21.1%)
Particular query workloads (i.e. Query workloads × *Design_customized*)	743	293 (39.4%)
Query workloads × DBD_basic	652	282 (43.3%)
Query workloads × DBD_fk	315	75 (23.8%)
Query workloads × DBD_customized	151	36 (23.8%)

Table 7.1: Overall results on *cluster1*. The percentages are in the relation to the *Total* column.

At the beginning of the tool run, the query bucket consisted of 66 *query-schema sets*[4]. This number is the combination of 22 query workloads (queries from the configuration file) and three tested schemata (*DBD_basic*,

4. *Query-schema sets* with *Desing_customized* schema are not counted since Database Designer does not work with this schema.

DBD_fk and *DBD_customized*). Only nine *query-schema sets* out of these 66 were non-optimal and were further investigated for that reason. Actually, these *query-schema sets* included only five unique query workloads – 4, 5, 18, 21, 23. These query workloads that involved only one query from the configuration file are hereinafter referred to as the *"base query workloads"*. Table 7.2 contains information about a distribution of non-optimal *query-schema sets* into particular *base query workloads* together with their tested schemata (*DBD_basic*, *DBD_fk*, and *DBD_customized*) and size of query workloads (1-tuple, 2-tuple and 3-tuple). The results show that mainly Queries *5* and *21* were challenging for Database Designer since both queries as *base query workloads* were non-optimal on all three tested schemata.

TIMEOUTS

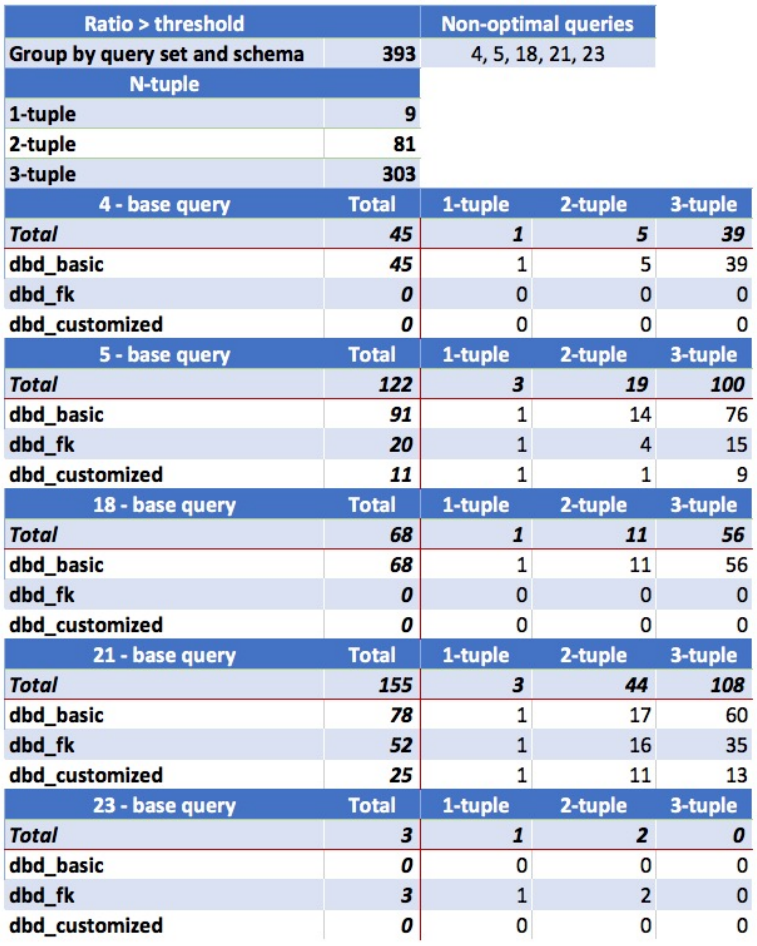
During the test, timeouts of queries arose in 69 cases. All timeouts occurred during the runs of Query *5* on *DBD_basic* schema, except one case related to run of Query *21* on *DBD_fk* schema. *Query-schema sets* affected by timeouts are listed in Appendix G on Page 101. To verify that the queries cannot complete their runs on given *query-schema sets*, the queries were also run manually on selected *query-schema sets*. Query *5* was successfully verified, since the runs of the query were not completed even after several hours. The manual run of Query *21* showed that this query was successfully completed. With data from snapshots of system tables, it was observed that the query was waiting 14 seconds for resources and the remaining time was not sufficient for its completion.

QUERY-SCHEMA SETS – ORDERED BY RATIO AND QUERY-RATIO

Measured data, which is contained in the attached Excel file, may be ordered by *ratio* and *query-ratio*[5] to show the worst cases for Database Designer. *Query-schema sets* whose ratio was higher than the threshold are listed in Appendix H on Page 102. The findings that are retrieved from *query-schema sets* ordered by ratio are:

- The worst 71 *query-schema sets* included only schema *DBD_basic*. Hereafter, also schemata *DBD_fk* and *DBD_customized* occurred.
- 60 *query-schema sets* with the highest response time had Query *5* as *base query workload*.

5. The ratio of one particular query from any query workload.



Ratio > threshold		Non-optimal queries
Group by query set and schema	393	4, 5, 18, 21, 23
N-tuple		
1-tuple	9	
2-tuple	81	
3-tuple	303	

4 - base query	Total	1-tuple	2-tuple	3-tuple
Total	***45***	***1***	***5***	***39***
dbd_basic	***45***	1	5	39
dbd_fk	***0***	0	0	0
dbd_customized	***0***	0	0	0

5 - base query	Total	1-tuple	2-tuple	3-tuple
Total	***122***	***3***	***19***	***100***
dbd_basic	***91***	1	14	76
dbd_fk	***20***	1	4	15
dbd_customized	***11***	1	1	9

18 - base query	Total	1-tuple	2-tuple	3-tuple
Total	***68***	***1***	***11***	***56***
dbd_basic	***68***	1	11	56
dbd_fk	***0***	0	0	0
dbd_customized	***0***	0	0	0

21 - base query	Total	1-tuple	2-tuple	3-tuple
Total	***155***	***3***	***44***	***108***
dbd_basic	***78***	1	17	60
dbd_fk	***52***	1	16	35
dbd_customized	***25***	1	11	13

23 - base query	Total	1-tuple	2-tuple	3-tuple
Total	***3***	***1***	***2***	***0***
dbd_basic	***0***	0	0	0
dbd_fk	***3***	1	2	0
dbd_customized	***0***	0	0	0

Table 7.2: Distribution of non-optimal *query-schema sets* among particular *base query workloads – cluster1*.

- The most non-optimal *query-schema set* consisted of the query workload with one Query *5*. The response time of the query was **13.05** times higher on *DBD_basic* than on *Design_customized*.
- The subsequent group of the most non-optimal *query-schema sets* had ratios from **5** to **7.3** and contained 12 query workloads composed of two queries, i.e. the tuples of queries had size 2.
- Group of *query-schema sets* that had ratios from **4** to **5** contained 47 query workloads composed of three queries, i.e. the tuples of queries had size 3.

- Clearly, from the previous three items, ratios decreased with increasing number of queries in *query workloads*. As can be expected, it was caused by a higher number of queries with optimal query-ratio that participated in the computation of total ratios of *query-schema sets*.
- The rest of *query-schema sets* had ratio lower than *4*.

Particular queries ordered by query-ratio are listed in the attached Excel file. The total number of run queries was 5,164; out of them, 1,155 queries (22.4%) were *non-optimal*. The most *non-optimal* query was Query *5* from query workload *(5, 10)* that run on *DBD_basic* schema.

INFLUENCE OF INFORMATION ON DATABASE DESIGNER OUTPUT

Based on Table 7.2, on *query-schema sets* order by ratio and on particular queries ordered by query-ratio, it is clear that Database Designer mostly creates optimal designs when referential integrity is included. The reason is that referential integrity helps the database engine understand relations among particular tables. If a segmentation clause is also included in the table definition, the optimality of proposals by Database Designer is generally even higher.

RESPONSE TIME, CPU TIME, USED MEMORY

DBPerfComp tool monitors the execution of every query of all query workloads. The monitoring includes retrieving *response time*, *CPU time*[6] and *used memory*[7] from system tables. This data is used for comparison of different schemata. Only *base query workloads* were selected for the comparison, since query workloads of two or three queries would improperly affect the relations among particular *base query workloads*. Figure 7.2 on Page 70 consists of three graphs, where each graph corresponds to one measured characteristics (response time, CPU time and used memory) and contains four bars, one for each schema. The findings retrieved from these graphs are:

6. A sum of CPU times spent in all threads. The CPU time is greatly influenced by disk access. The relation between CPU time and Response time mostly depends on the nature of given query and on current database/hardware load.
7. Memory that was used during a query execution.

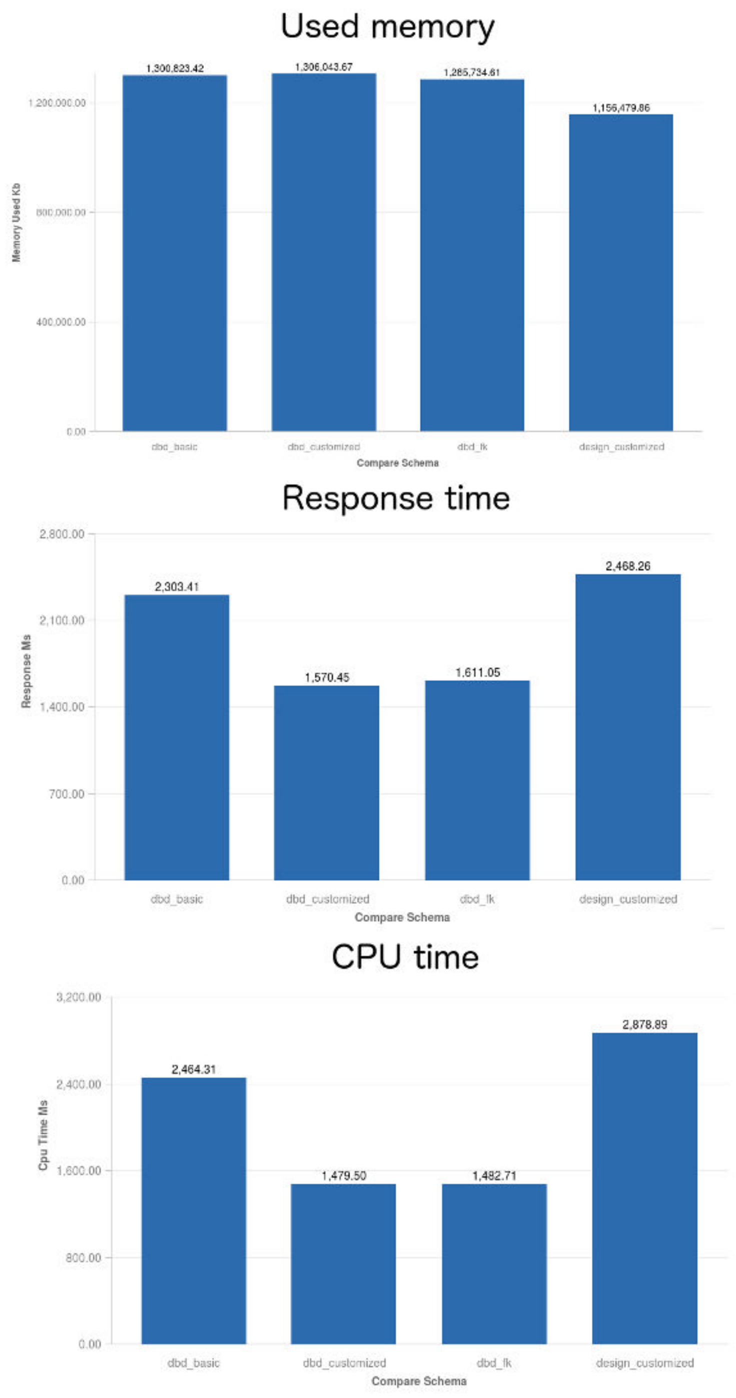


Figure 7.2: Graphs of used memory, response time and CPU time where query workloads composed of one query from the configuration file are counted in – *cluster1*.

- The highest amount of used memory occurs at schema *DBD_basic*. In contrast, the smallest amount occurs at schema *Design_customized*. These results show that more memory is required with higher complexity of projections (e.g. ordering, encoding).
- The graph with response time confirms that when referential integrity is added to the table definitions, the level of optimality of projections designed by Database Designer is markedly increased. Clearly, queries executed on schemata *DBD_fk* and *DBD_customized* needed twice less response time than on schema *DBD_basic*. The fact that Database Designer does not create optimal projections without stating referential integrity is supported with another fact that executed queries needed less response time even on *Design_customized* than on *DBD_basic*.
- The graph with CPU time has similar relations among particular schemata. For that reason, it confirms the higher level of optimality in the cases when referential integrity is included in table definitions.
- Database Designer significantly improved response time and CPU time at schemata *DBD_fk* and *DBD_customized*. On the other hand, schema *DBD_basic* was not successfully optimized.

Characteristics response time and CPU time are further examined in the graphs in Figures 7.3 and 7.4. In both the graphs, values are segmented by tested schemata (particular lines), and the horizontal axis represents particular *base query workloads*. The findings retrieved from these graphs are:

- At most query workloads, schema *Design_customized* is less optimal than all schemata designed by Database Designer.
- Except the workload with Query *21*, both graphs of response time and CPU time are similar.
- Schema *Design_customized* is significantly more optimal only at singleton *21*. But, this fact could have been caused only by un unexpected load in database. As snapshots of system tables showed, the reason for this possible explanation is that executed Query *21* was waiting on resources for a long time in the case of run on schemata *DBD_basic*, *DBD_fk* and *DBD_customized*. Exact database load in time of the query run is not possible to get since only snapshots of data related to DBPerfComp tool were maintained.

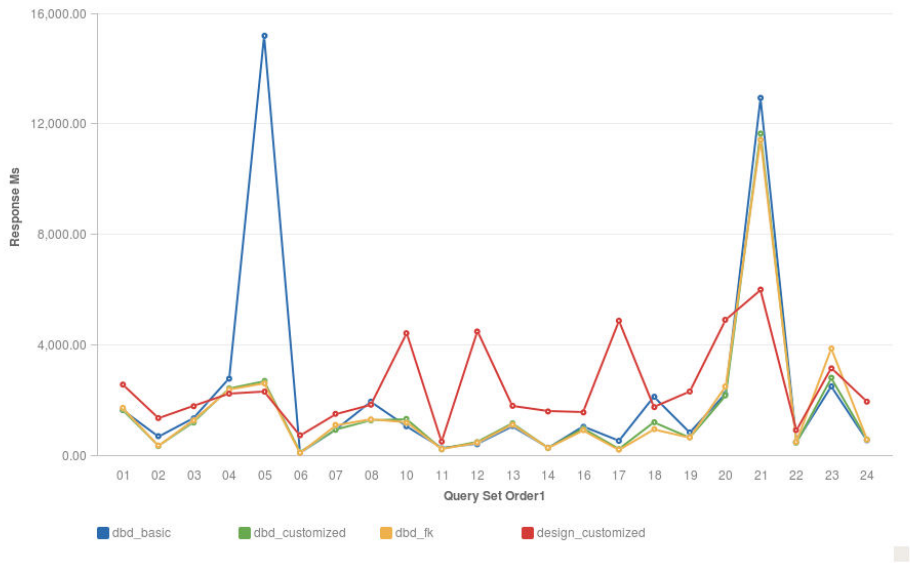


Figure 7.3: Response time for particular *base query workloads* and tested schemata – *cluster1*.

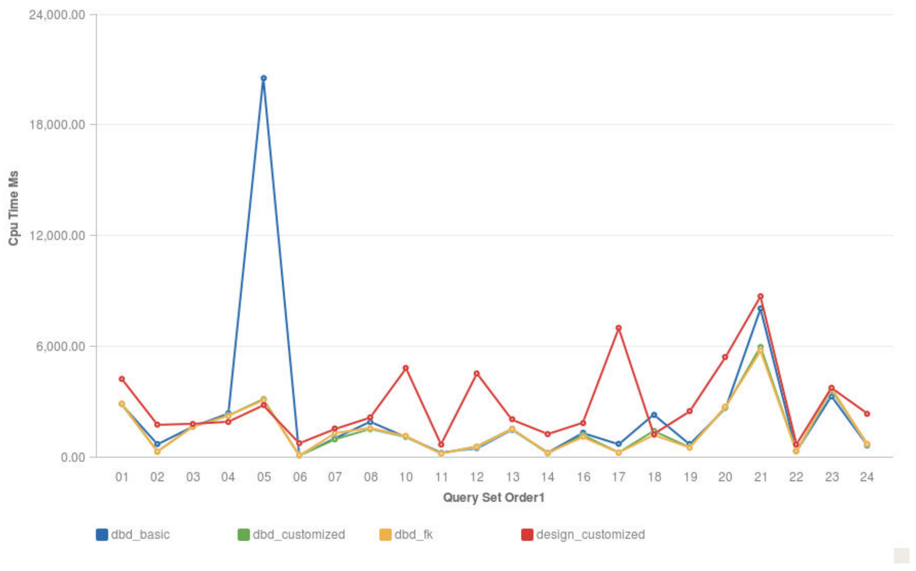


Figure 7.4: CPU time for particular *base query workloads* and tested schemata – *cluster1*.

Time of designing process

Times of designing processes are compared in the graph in Figure 7.5. Apparently, Database Designer requires more time in the case of no referential integrity at tables that participate in the design. The appearance of segmentation did not have any influence, since times of designing processes at schemata *DBD_fk* and *DBD_customized* were equally changing.

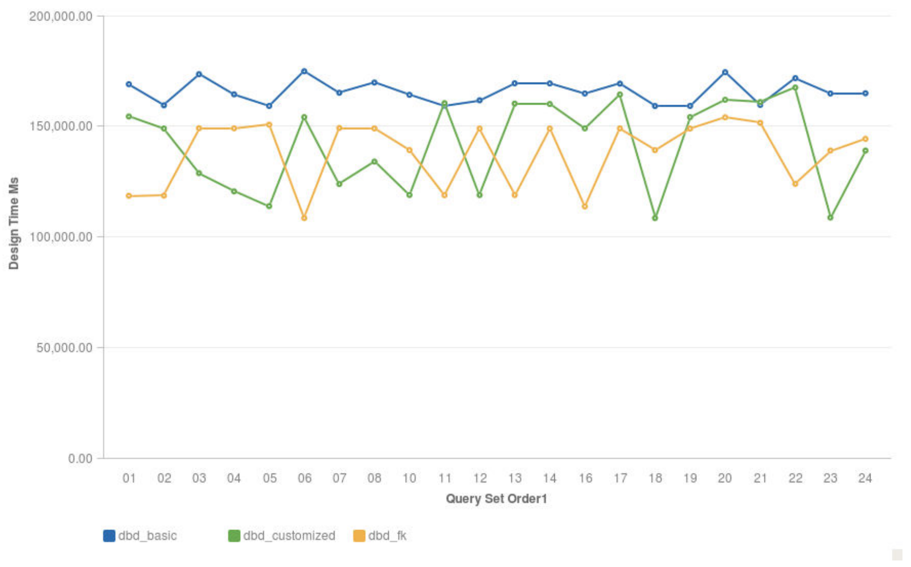


Figure 7.5: Design time for particular query workloads composed of one query from the configuration file and tested schemata – *cluster1*.

7.2.2 Cluster 2

This section contains data from measurements on *cluster2*. Since many timeouted queries occurred at *cluster1*, the timeout was set to 600 seconds to get more precise information about running these time-consuming queries. An overview of results on *cluster2* is shown in Table 7.3 that has the same structure as Table 7.1.

With comparison of the measurements on *cluster1* (Table 7.1) and *cluster2* (Table 7.3), it is clear that the results were similar on both clusters. *Cluster1* results contained 1,861 query-schema sets where 393 of them had ratio higher than threshold (21,1%). To compare, *cluster2* results contained 1,981 query-schema sets where 336 of them had ratio higher than threshold (17%). With further investigation, all values and relations between re-

	Total	**Ratio > Threshold**
Combinations: Query workload × Schema (with *Design_customized*)	1,981	**336** (17.0%)
Particular query workloads (i.e. Query workloads × *Design_customized*)	640	241 (37.6%)
Query workloads × DBD_basic	600	196 (32.7%)
Query workloads × DBD_fk	395	75 (19.0%)
Query workloads × DBD_customized	346	65 (18.8%)

Table 7.3: Overall results on *cluster2*. The percentages are in the relation with the *Total* column.

lated parts are similar in both measurements with one exception at schema *DBD_customized*. There, measurement on *cluster2* includes approximately twice more *query-schema sets*.

Measurements on *cluster2* included ten non-optimal combinations of *base query workloads* and tested schemata *DBD_basic*, *DBD_fk*, and *DBD_customized*. The unique non-optimal *base query workloads* were composed of Queries *4*, *5*, *11* and *18*. Common non-optimal *base query workloads* for both clusters were with Queries *4*, *5* and *18*. The complete distribution of query-schema sets among particular *base query workloads* is shown in Appendix I in Table I.1 on Page 104.

In the measurement on *cluster2*, a timeout occurred at 42 queries that involved Query *5* and schema *DBD_basic* in all cases.

Query-schema sets order by ratio and particular queries ordered by ratio-query are listed in the attached Excel file.

Particular graphs from *cluster2* contain similar trends as results from *cluster1*. The graphs are shown in Appendix I on Pages 105, 106 and 107.

Because of the similarity with *cluster1*, next sections in this chapter involve only measurements from that cluster.

7.3 Query 5

This section provides the description of Query *5*, since the results in Section 7.2 show that it the most challenging query for Database Designer. The definition of Query *5* is as follows:

```
SELECT
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
FROM
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
WHERE
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'MIDDLE EAST'
        and o_orderdate >= date '1997-01-01'
        and o_orderdate < date '1997-01-01' + interval '1' year
GROUP BY
        n_name
ORDER BY
        revenue desc
LIMIT 1;
```

The description of Query *5* from the official TPC-H specification[24] is:

> *"This query lists the revenue volume done through local suppliers. (...) The Local Supplier Volume Query lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. The query is run in order to determine whether to institute local distribution centers in a given region. The query considers only parts ordered in a given year. The query displays the nations and revenue volume in descending order by revenue. Revenue volume for all qualifying lineitems in a particular nation is defined as sum(l_extendedprice * (1 - l_discount))."*

In comparison to other TPC-H queries, this query is specific in the cohesion of joins that is shown in Figure 7.6. There, tables *Supplier*, *Lineitem*, *Orders*, and *Customers* are joined mutually as it is shown with the blue, red, green and yellow lines.

Query *5* works with two largest tables – *Lineitem* and *Orders*. As it is clear from Figure 7.6, each table participate in two *join* operations. At both tables, one of the operations joins these two largest tables together.

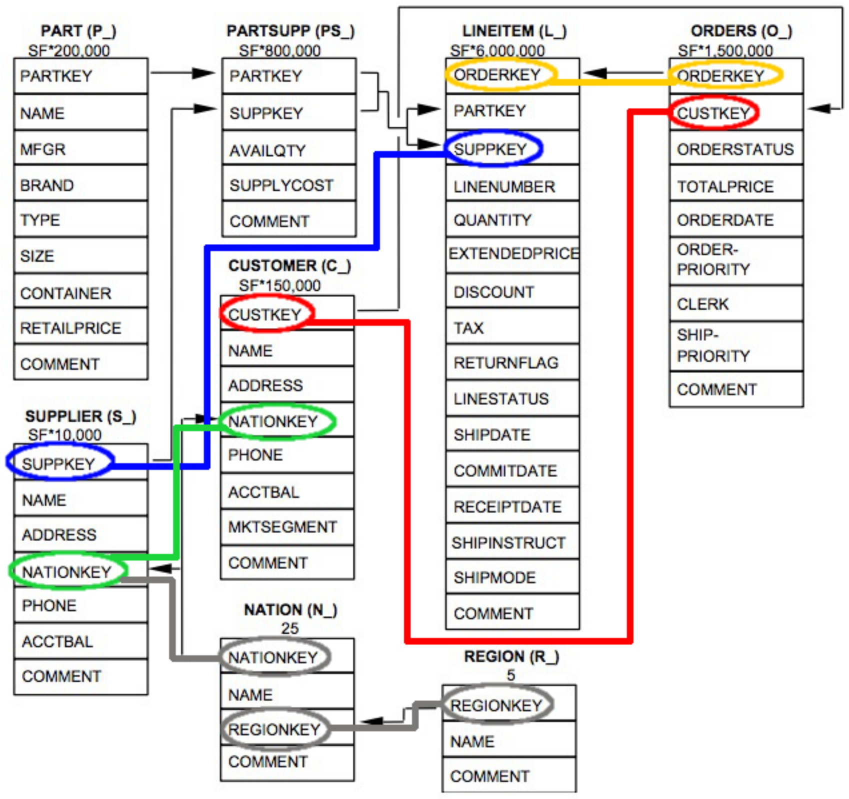


Figure 7.6: Illustrated joins of Query 5 in TPC-H schema.

Based on the two previous paragraphs, it is apparent that *join* operations are the challenging part of Query 5. The query is further investigated in Sections 7.4, 7.5 a 7.6.

7.4 Projection definition

This section explores definitions of projections that were designed by Database Designer with optimization for Query 5 on schema *DBD_basic*. More precisely, the projections were optimized for query workload (*5*, *10*) since it involved the highest query-ratio; i.e. the least optimal single query (Query *5*) run on tables that included no referential integrity and no segmentation definitions, and the tables were optimized for Queries *5* and *10*. For this *query-schema set*, the output of Database Designer contains ten following projections and their segmentation:

- *Region* – unsegmented (replicated),
- *Nation* – unsegmented (replicated),
- *Supplier* – unsegmented (replicated),
- *Part* – segmented by NAME,
- *Partsupp* – segmented by COMMENT,
- *Customer* – – segmented by CUSTKEY,
- *Orders 1* – segmented by CUSTKEY (used for Query *5*),
- *Orders 2* – segmented by ORDERKEY (used for Query *10*),
- *Lineitem 1* – segmented by SUPPKEY (used for Query *5*),
- *Lineitem 2* – segmented by ORDERKEY (used for Query *10*).

From the findings that were retrieved during all measurements[8], it is apparent that Database Designer uses *join* statements to create a segmentation clauses of particular projections. Besides, Database Designer replicates small tables over all nodes; here, it is the case of Region, Nation and Supplier tables. Both of the largest tables – Lineitem and Orders – involve two projection where each projection is optimized for a different query. Furthermore, based on designed projections (attached in the Excel file), it is apparent that Database Designer optimizes columns ordering for *WHERE* clauses. Then, no general pattern was found for the columns that follow the ones that are optimized for *WHERE* clause – they were optimized for one of the cardinality, join columns and *GROUP BY* clauses.

Each *query-schema set* that involved Query *5* with timeout included the *l_suppkey* column at the first position of segmentation clause in Lineitem table. On the other hand, only two *query-schema sets* that involved Query *5*, and *l_suppkey* column at the first position of segmentation clause finished without a timeout. In the query, the *l_suppkey* column is joined with the primary key of Supplier table.

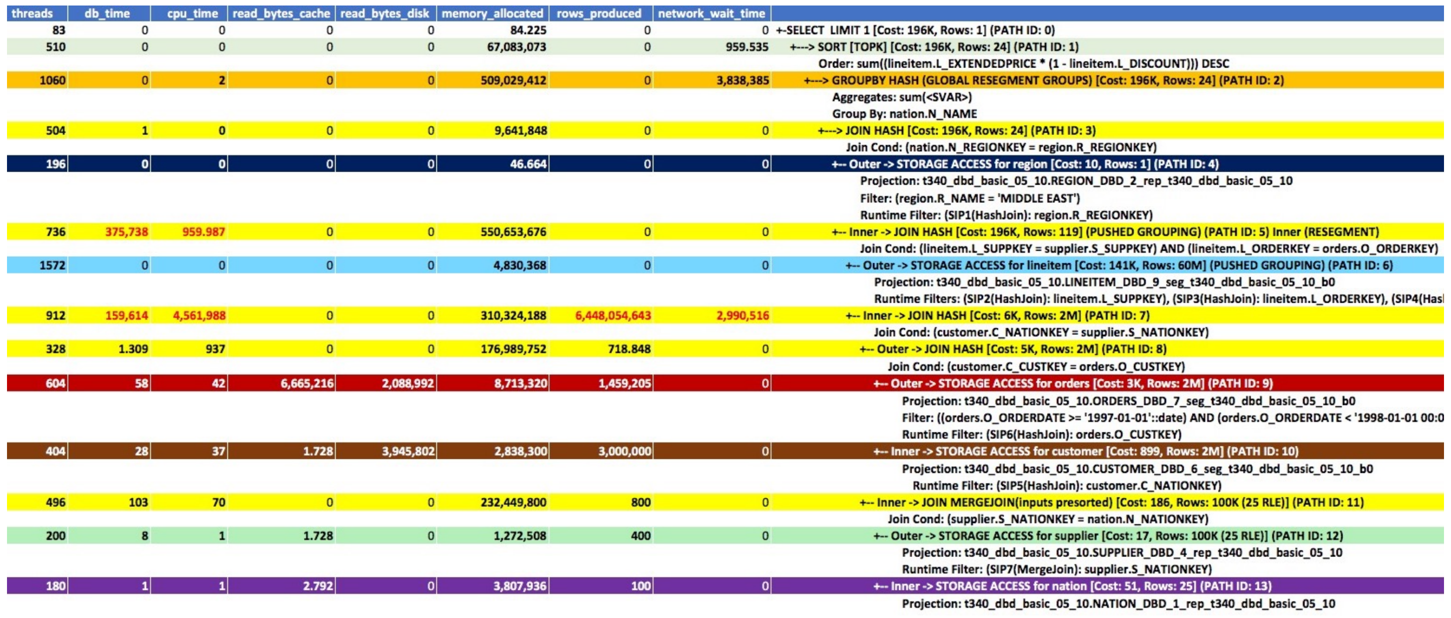
8. The attached Excel file contains examples of designed projections for different *query-schema sets* (including designed projections for query workload (*5*, *10*) and schema *DBD_basic*).

7.5 Query plan

Query plan helps understand precise steps performed during the query execution; in this case, the execution of Query *5*. The selected *query-schema set* for exploration is the same as in the previous section – query workload *5, 10* and schema *DBD_basic*. The investigation query, shown in Appendix J on Page 108, enables retrieving data of many characteristics for each step of the query plan; for example *database time*, *CPU time*, *network wait time*, *allocated memory* and *number of produced rows*. This investigation of the query plan was performed after the whole DBPerfComp tool run because of the possibility to observe changes of values in particular steps of the query plan. As the query was stuck and was not able to finish, it was manually stopped after some time.

Retrieved query plan with measured characteristics for each step is presented in Figure 7.7 in shortened version. The full version is present in the attached Excel file, sheet Query plan. Red values in particular steps in Figure 7.7 signify values that were being increased with time of running query. The findings are:

- First, storage accesses are performed for tables Nation, Supplier, Customer, and Orders.
- Then, joins Customer $\times$ Orders (*path id* = 8) and Nation $\times$ Supplier (*path id* =11) are executed.
- The following *join* (*path id* = 7) makes *join* of these two *joins* (*path id = 8, 11*). The query is stuck on this *join* (*path id* = 7), since *DB time*, *CPU time*, *Rows produced* and *Network wait time* are being continually increased. The possible explanation for this behavior is that the joined tables are not optimally prepared (segmentation, ordering) for this *join* (*path id* = 7); for that reason, the *join* (segmentation, ordering) is running for undefined time.
- Simultaneously, *join* (*path id* = 7) sends data to the next operator that is *join* with *path id* 5. Because the previous *join* (*path id* = 7) runs for undefined time, also the *join* with *path id* 5 cannot be finished, since it receives data all that time.
- Operators at *path id* 4 or smaller are not to be performed, because they only wait for previous operators.



threads	db_time	cpu_time	read_bytes_cache	read_bytes_disk	memory_allocated	rows_produced	network_wait_time	
83	0	0	0	0	84.225	0	0	+-SELECT LIMIT 1 [Cost: 196K, Rows: 1] (PATH ID: 0)
510	0	0	0	0	67,083,073	0	959.535	+---> SORT [TOPK] [Cost: 196K, Rows: 24] (PATH ID: 1)
								Order: sum((lineitem.L_EXTENDEDPRICE * (1 - lineitem.L_DISCOUNT))) DESC
1060	0	2	0	0	509,029,412	0	3,838,385	+---> GROUPBY HASH (GLOBAL RESEGMENT GROUPS) [Cost: 196K, Rows: 24] (PATH ID: 2)
								Aggregates: sum(<SVAR>)
								Group By: nation.N_NAME
504	1	0	0	0	9,641,848	0	0	+---> JOIN HASH [Cost: 196K, Rows: 24] (PATH ID: 3)
								Join Cond: (nation.N_REGIONKEY = region.R_REGIONKEY)
196	0	0	0	0	46.664	0	0	+-- Outer -> STORAGE ACCESS for region [Cost: 10, Rows: 1] (PATH ID: 4)
								Projection: t340_dbd_basic_05_10.REGION_DBD_2_rep_t340_dbd_basic_05_10
								Filter: (region.R_NAME = 'MIDDLE EAST')
								Runtime Filter: (SIP1(HashJoin): region.R_REGIONKEY)
736	375,738	959.987	0	0	550,653,676	0	0	+-- Inner -> JOIN HASH [Cost: 196K, Rows: 119] (PUSHED GROUPING) (PATH ID: 5) Inner (RESEGMENT)
								Join Cond: (lineitem.L_SUPPKEY = supplier.S_SUPPKEY) AND (lineitem.L_ORDERKEY = orders.O_ORDERKEY)
1572	0	0	0	0	4,830,368	0	0	+-- Outer -> STORAGE ACCESS for lineitem [Cost: 141K, Rows: 60M] (PUSHED GROUPING) (PATH ID: 6)
								Projection: t340_dbd_basic_05_10.LINEITEM_DBD_9_seg_t340_dbd_basic_05_10_b0
								Runtime Filters: (SIP2(HashJoin): lineitem.L_SUPPKEY), (SIP3(HashJoin): lineitem.L_ORDERKEY), (SIP4(Has
912	159,614	4,561,988	0	0	310,324,188	6,448,054,643	2,990,516	+-- Inner -> JOIN HASH [Cost: 6K, Rows: 2M] (PATH ID: 7)
								Join Cond: (customer.C_NATIONKEY = supplier.S_NATIONKEY)
328	1.309	937	0	0	176,989,752	718.848	0	+-- Outer -> JOIN HASH [Cost: 5K, Rows: 2M] (PATH ID: 8)
								Join Cond: (customer.C_CUSTKEY = orders.O_CUSTKEY)
604	58	42	6,665,216	2,088,992	8,713,320	1,459,205	0	+-- Outer -> STORAGE ACCESS for orders [Cost: 3K, Rows: 2M] (PATH ID: 9)
								Projection: t340_dbd_basic_05_10.ORDERS_DBD_7_seg_t340_dbd_basic_05_10_b0
								Filter: ((orders.O_ORDERDATE >= '1997-01-01'::date) AND (orders.O_ORDERDATE < '1998-01-01 00:0
								Runtime Filter: (SIP6(HashJoin): orders.O_CUSTKEY)
404	28	37	1.728	3,945,802	2,838,300	3,000,000	0	+-- Inner -> STORAGE ACCESS for customer [Cost: 899, Rows: 2M] (PATH ID: 10)
								Projection: t340_dbd_basic_05_10.CUSTOMER_DBD_6_seg_t340_dbd_basic_05_10_b0
								Runtime Filter: (SIP5(HashJoin): customer.C_NATIONKEY)
496	103	70	0	0	232,449,800	800	0	+-- Inner -> JOIN MERGEJOIN(inputs presorted) [Cost: 186, Rows: 100K (25 RLE)] (PATH ID: 11)
								Join Cond: (supplier.S_NATIONKEY = nation.N_NATIONKEY)
200	8	1	1.728	0	1,272,508	400	0	+-- Outer -> STORAGE ACCESS for supplier [Cost: 17, Rows: 100K (25 RLE)] (PATH ID: 12)
								Projection: t340_dbd_basic_05_10.SUPPLIER_DBD_4_rep_t340_dbd_basic_05_10
								Runtime Filter: (SIP7(MergeJoin): supplier.S_NATIONKEY)
180	1	1	2.792	0	3,807,936	100	0	+-- Inner -> STORAGE ACCESS for nation [Cost: 51, Rows: 25] (PATH ID: 13)
								Projection: t340_dbd_basic_05_10.NATION_DBD_1_rep_t340_dbd_basic_05_10

Figure 7.7: Part of the query plan for Query *5* in query workload *05_10* and in schema *DBD_basic*.

This investigation confirmed that the issue of unfinished queries is caused by *join* operations. *Join* with *path id* 7 has to reorder large table where all records from Orders table occur. Furthermore, *join* with *path id* 5 needs to execute resegmentation on data that is joined in this step. Thus, this new large table of four tables (Orders, Customer, Supplier, Nation) is created, which is then joined (*path id* = 5) with the largest table Lineitem. This join (path id = 5) is specific since the join condition includes two parts:

(lineitem.L_SUPPKEY = supplier.S_SUPPKEY) AND
(lineitem.L_ORDERKEY = orders.O_ORDERKEY)

The most possible explanation of this Query *5* issue corresponds to the following structure of *joins*:

t1.a = t2.a *AND t2.b = t3.b AND t3.c = t4.c AND* ***t1.d = t4.d***

This explanation was further investigated with the simplified version of Query *5* that is shown in Appendix K on Page 110. In contrast to traditional Query *5*, this simplified query does not contain tables Nation and Region that are not involved in the critical structure of *joins*. Moreover, *GROUP BY* clause is also skipped. This query was run on the projections optimized for query workload (*5, 10*) and schema *DBD_basic*. According to the result, a timeout also occurred at the run of the simplified version of Query *5*. This measurement confirmed the previous possible explanation.

Vertica does not enable finding more precise information about these *joins* and the exact location of the issue. The possible explanation is that Vertica performs cross *join*, but this operation is beyond Vertica's resources.

7.6 Database operations

This section investigates operations that are performed within a run of Query *5*. The operations (operators and their counters) are stored in *execution_engine_profiles* table and are mostly described in [22] and in the Vertica official documentation[8], chapter *Execution Engine Profiles*. The table *execution_engine_profiles* holds information about the precise execution of queries – particular operators and their counters with values. System table *dc_requests_issued* was also used to identify the executed query.

The investigated data was retrieved with monitoring query that is shown in Appendix L on Page 111. The retrieved data contains *operator name*, *counter name*, *duration (µs)*, *number of threads* and *duration (µs) per thread*.

This monitoring was performed on ten queries (*query-schema sets*) with the highest query-ratio; these all queries were Queries *5*. All results of this monitoring are stored in the attached Excel file. All ten measurements retrieved the same operators and their counters. Table 7.4 presents averages of measured values. The table is divided into four parts:

1. Results on tested schema from *query-schema set*.
2. Results on *Design_customized* schema (base schema).
3. The difference between results from these two schemata (Tested schema – *Design_customized*). The higher positive difference, the worse design of tested schema.
4. Ratio between results from these two schemata. (*Design_customized* / Tested schema). The lower ratio, the worse design of tested schema.

The findings retrieved from Table 7.4 are:

- *NetworkSend* and *NetworkRecv* were operators with the highest durations. The first operator represents data to be sent to another node in cluster and the second operator represents data to be received from another node.
- Operator *Join* is impossible to assign to one particular join when the query contains more joins than one. The reason is that durations of all joins are summed to one value. In this investigation, it is obvious that the joins are significant for query execution.
- *AVG ratio* part and its *Duration (μs)* column show that schema *Design_customized* is much more optimal for Query *5* than schema *DBD_basic*. In most of the measured operators and counters, values for *Design_customized* are at least 60% better than values for *DBD_basic*.

The attached Excel file also contains data from other investigations of performed operations with Queries *10, 12, 13, 14, 18, 21, 22* and *24*. These queries participate in *query-schema sets* with high query-ratio. The results of these queries are not further described, but the general usage of particular operators is concluded from these measurements. In general, results from all investigations show that operators and their counters with the highest durations are[9]:

9. Descriptions of particular counters are in Vertica documentation[8], chapter Execution Engine Profiles

AVG of top 10 worst query executions - DBD_basic schema					Design_customized schema			AVG difference - DBD_basic - Design_customized					AVG ratio = Design_customized / DBD_basic		
Operator name	Counter name	Duration (us)	Threads	Duration (us) per thread	Duration (us)	Threads	Duration (us) per thread	Operator name	Counter name	Duration (us)	Threads	Duration (us) per thread	Duration (us)	Threads	Duration (us) per thread
NetworkSend	consumer stall (us)	893786	720	124136	123933	600	20656	NetworkSend	consumer stall (us)	769853	120	103480	0.139	0.833	0.166
NetworkRecv	network wait (us)	724173	540	134105	91702	450	20378	NetworkRecv	network wait (us)	632471	90	113727	0.127	0.833	0.152
Join	join outer clock time	510198	328	155827	67495	960	7031	Join	join outer clock time	442703	-632	148796	0.132	2.927	0.045
ParallelUnion	response wait (us)	354498	160	221561	102340	480	21321	ParallelUnion	response wait (us)	252158	-320	200240	0.289	3.000	0.096
NetworkSend	producer wait (us)	300813	240	125338	50158	240	20900	NetworkSend	producer wait (us)	250655	0	104438	0.167	1.000	0.167
StorageUnion	response wait (us)	238267	440	54150	52852	840	6293	StorageUnion	response wait (us)	185415	-400	47857	0.222	1.909	0.116
Join	join inner clock time	235155	488	48301	27987	960	2915	Join	join inner clock time	207168	-472	45386	0.119	1.967	0.060
ParallelUnion	clock time (us)	173186	160	108242	30565	480	6369	ParallelUnion	clock time (us)	142621	-320	101873	0.176	3.000	0.059
Join	clock time (us)	138543	488	28542	37418	960	3900	Join	clock time (us)	101125	-472	24642	0.270	1.967	0.137
Join	join outer execution	137680	328	42650	46712	960	4866	Join	join outer execution	90968	-632	37784	0.339	2.927	0.114
Join	execution time (us)	58667	488	12126	20311	960	2116	Join	execution time (us)	38356	-472	10010	0.346	1.967	0.175
NetworkSend	execution time (us)	45285	229	19711	0	243	0	NetworkSend	execution time (us)	45285	-14	19711	0.000	1.061	0.000
Join	join inner execution	35913	488	7363	8840	960	920	Join	join inner execution	27073	-472	6443	0.246	1.967	0.125
NetworkRecv	execution time (us)	27802	180	15447	6	150	1	NetworkRecv	execution time (us)	27796	30	15446	0.000	0.833	0.000
Filter	clock time (us)	21693	728	2980	1260	1560	80	Filter	clock time (us)	20433	-832	2900	0.058	2.143	0.027
Root	input queue wait (us)	20851	10	208518	6809	30	22696	Root	input queue wait (us)	14042	-20	185822	0.327	3.000	0.109
Filter	execution time (us)	16626	728	2285	1179	1560	79	Filter	execution time (us)	15447	-832	2206	0.071	2.143	0.035
NetworkSend	producer stall (us)	11888	80	14860	0	456	0	NetworkSend	producer stall (us)	11888	-376	14860	0.000	5.700	0.000
StorageUnion	execution time (us)	6399	439	1457	156	834	19	StorageUnion	execution time (us)	6243	-395	1438	0.024	1.900	0.013
NetworkSend	network wait (us)	5809	240	2421	1	600	0	NetworkSend	network wait (us)	5808	-360	2421	0.000	2.500	0.000
Join	join inner hash table building time (us)	586	204	293	8719	480	1818	Join	join inner hash table building time (us)	-8133	-276	-1525	14.879	2.353	6.205
Scan	clock time (us)	488	440	111	15000	1080	1388	Scan	clock time (us)	-14512	-640	-1277	30.738	2.455	12.505
Scan	execution time (us)	376	460	84	11720	1080	1086	Scan	execution time (us)	-11344	-620	-1002	31.170	2.348	12.929
NewEENode	initialization time (us)	263	80	329	482	120	402	NewEENode	initialization time (us)	-219	-40	-73	1.833	1.500	1.222
NetworkRecv	initialization time (us)	181	180	100	186	150	124	NetworkRecv	initialization time (us)	-5	30	-24	1.028	0.833	1.240

Table 7.4: The investigation of operations that are performed within a run of Query 5. The values are averages of ten runs of Query 5 on schemata from *query-schema sets* with the highest query-ratio.

- *NetworkSend* – counters *producer wait* and *consumer stall*
- *NetworkRecv* – counter *network wait,*
- *Join* – counters *join inner clock time* and *join outer clock time,*
- *ParralelUnion*[10] – counter *response wait,*
- *StorageUnion*[11] – counter *response wait*.

This type of investigation provided an overview of particular operations that are performed within the execution of Query *5*. The most time-consuming actions were connected to the network and join operations. These results correspond to the investigation of query plan in the previous section, where it was observed that join operations are very challenging at Query *5* and that large volume of data had to be sent over a network due to segmentation. If the measurements were not performed on a cluster, but on a single machine, the results would be different.

7.7 Manual optimization

All *query-schema sets* that involved a timeout contained *l_suppkey* column at the first position of the segmentation clause in Lineitem table, as it was already described in Section 7.4; this section also listed columns from segmentation clauses of the other tested tables. Section 7.7 explores possible manual optimizations of the Lineitem projection that was originally optimized[12] for Query *5*. The two proposed optimizations are:

1. Change segmentation of Lineitem projection from *l_suppkey* to *l_orderkey* – according to the join *"l_orderkey = o_orderkey"*.
2. Change segmentation of Lineitem projection from *l_suppkey* to *l_extendedprice* – according to the *"l_orderkey = o_orderkey"*.

10. *"Combines not sorted data stream.*[22]
11. *"Combines data storage without maintaining the sort order."*[22]
12. More precisely, the projections were optimized for query workload (*5*, *10*) and schema *DBD_basic*; but Section 7.7 explores only projections that are actually used by Query *5*. Projections used by Query *10* are not covered in the section.

Both optimizations led to successful completion of Query 5. More effective optimization was the first one (*l_orderkey*) since smaller response time (3,021 milliseconds × 19,461 milliseconds) and CPU time (4,033 milliseconds × 7,565 milliseconds) was measured with that optimization. Complete results of these two measurements are situated in the attached Excel file. This measurement showed that *l_suppkey* column at the first position of the segmentation clause in Lineitem table affects the timeouts of Query 5.

7.8 Results summary

Measurements on both clusters showed how Database Designer optimizes projections for various query workloads. According to the results, different amount of information (segmentation, referential integrity) in tables definitions greatly affects the level of optimization. Based on the results, it was observed that referential integrity in tables definition helps the database engine understand data model relations and therefore optimize join operations. In performed measurements, the inclusion of the referential integrity to table definition highly increased the level of optimization. Including additional segmentation clauses in table definitions influenced the results only to a small degree.

Tests on both clusters measured 111 timeouted queries. Except for one case, all timeouts occurred at Query 5 and schema *DBD_basic*. The total number of tested query-schema sets (without *Design_customized* schemata) is 2,459 where 729 out of them (29.6%) were non-optimized – 478 at *DBD_basic*, 150 at *DBD_fk*, and 101 *DBD_customized*. These numbers confirm that Database Designer creates more optimal projections when tables definitions contain referential integrity and segmentation clause.

The worst cases for Database Designer were query workloads that included Query 5 and schema *DBD_basic*. Based on all measurements, Query 5 is the outlier for Database Designer. Despite the fact that Vertica provides detail query profiling, it enables localizing only the root cause of an issue in query plan, and in some cases, localizing a certain database operator. At the combination of Query 5 and schema *DBD_basic*, it is clear that join operations of the query cause the non-ability to finish its run. As the only query of TPC-H model, Query 5 involves four tables *Supplier*, *Lineitem*, *Orders*, and *Customers* that are mutually joined as it is shown in Figure 7.6. The structure of these *joins* is shown below:

t1.a = t2.a *AND t2.b = t3.b AND t3.c = t4.c AND* ***t1.d = t4.d***

The most possible explanation is that this join distribution causes the issues. Moreover, every time a timeout occurred, the segmentation of table Lineitem included column *l_suppkey* at the first position. It was explored that Query *5* successfully completes its run when *l_suppkey* column in the segmentation clause is replaced with *l_orderkey* or *l_extendedprice* columns.

Due to the investigation of database operators, it was observed that the most resource-consuming operators are *NetworkSend*, *NetworkRecv*, *Join*, *ParralelUnion*, and *StorageUnion*.

Results of this chapter prove that Database Designer does not always work optimally, because many designed schemata had worse results than schema that served as the input for the designing process. If designs of Database Designer were always optimal, all measured ratios would be smaller than 1.

8 Conclusion

Column-oriented database systems offer many benefits in the case of analytics workloads, since queries work only with required columns and the others are skipped. The major improvements are gained for read-only queries, especially if an aggregation function is included. With fast-growing volumes of data, data analytics are of higher importance. In contrast, row-oriented systems hold their benefits in the case of transactional workloads where column-oriented systems are highly inefficient.

Vertica is relational column-oriented database system that provides many specific features. The most characteristic one is projections that represents a physical schema of data. The main benefit of projections is that they may be optimized for given query workloads. Vertica includes built-in Database Designer tool that performs this optimization with usage of ordering, segmentation, selection and compression method for particular columns.

All measurements in this thesis used TPC-H model. The analysis of locking subsystem (delays due to locks) proved that Database Designer does not influence database availability. The measured data showed that Database Designer creates a vast number of locks, but their average of being acquired is only a few milliseconds. Moreover, the total sum of the delays represents a small fraction of the total time of the designing process of Database Designer. The results indicate that Database Designer might be used in a critical database environment.

The main contribution of this thesis is the implemented DBPerfComp tool that targets suboptimal designs generated by Database Designer. The measured data shows that Database Designer generates many non-optimal designs. The results prove that an amount of information in the definitions of tables whose projections are optimized influences the resulting optimality of designed projections. Within the test, one combination of the query and the schema was detected as critical for Database Designer. This combination includes Query *5* from TPC-H model and the schema that involves no segmentation definition and no referential integrity in the definitions of its tables. Unfortunately, Vertica does not provide sufficient information to find the exact cause of this issue. The possible explanation is that Vertica performs cross join that is not able to complete due to the huge tables. To verify these results, the tool was also executed on the different cluster. The results obtained are very similar to the previous ones.

DBPerfComp tool has already been used in the GoodData s.r.o. company. Based on the uncovered issue that was described in the previous paragraph, the official bug report was created by GoodData s.r.o. and on April 28, 2017, sent to Vertica. This bug was accepted with notification that developers in Vertica would further examine such Database Designer behavior. The official bug report created by GoodData s.r.o. is included in Appendix M on Page 112. DBPerfComp may be further used with new releases of Vertica to test whether current issues are repaired and to find new possible issues.

This thesis offers three possibilities of future work. The first option is to investigate all doubles and triples without restriction on those combinations of query workloads and schemata that are not designed optimally. The second option is to include machine learning that would take measured data from different machines and different volumes of data. Then, needed information (specification, dependencies) would be added, and the machine learning would enable prediction of Database Designer behavior for given hardware configuration. The third option is to extend the implemented tool in the way that for the non-optimized design, the tool would create a new optimized design with a usage of data statistics.

Bibliography

[1] Sezin G. Yaman. Introduction to column-oriented database systems. In *Seminar: Columnar Databases*. University of Helsinky, November 2012.

[2] Db-engines ranking, 2017. `https://db-engines.com/en/ranking`. Datum: 13. 4. 2017.

[3] TimeStored. The top column-oriented databases compared, 2017. `http://www.timestored.com/time-series-data/column-oriented-databases`. Datum: 18. 4. 2017.

[4] HPE Vertica. Architecture overview, 2017. `https://my.vertica.com/get-started-vertica/architecture/`. Datum: 13. 4. 2017.

[5] Gheorghe Matei. Column-Oriented Databases, an Alternative for Analytical Environment. *Database Systems Journal*, 1(2):3–16, December 2010.

[6] Vandana Bhagat and Arpita Gopal. Comparative study of row and column oriented database. In *2012 Fifth International Conference on Emerging Trends in Engineering and Technology*, pages 196–201, Nov 2012.

[7] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[8] HPE Vertica. Vertica documentation – software version: 8.0.x, 2017. `https://my.vertica.com/docs/8.0.x/PDF/Vertica_8.0.x_Complete_Documentation.pdf`. Datum: 3. 4. 2017.

[9] Jim Knicely. Global catalog and local catalog, 2017. `http://vertica-forums.com/viewtopic.php?t=2331`. Datum: 21. 4. 2017.

[10] David Portnoy. Comparison of mpp data warehouse platforms, 2017. `https://www.slideshare.net/DavidPortnoy/comparison-of-mpp-data-warehouse-platforms`. Datum: 18. 4. 2017.

[11] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *CoRR*, abs/1208.4173, 2012.

[13] Vertica Tips. Release history, 2017. `http://vertica.tips/release-history/`. Datum: 18. 4. 2017.

[14] Nga Tran, Andrew Lamb, Lakshmikant Shrinivas, Sreenath Bodagala, and Jaimin Dave. The vertica query optimizer: The case for specialized query optimizers. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1108–1119, March 2014.

[15] Ramakrishna Varadarajan, Vivek Bharathan, Ariel Cary, Jaimin Dave, and Sreenath Bodagala. Dbdesigner: A customizable physical design tool for vertica analytic database. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1084–1095, March 2014.

[16] HPE Vertica. Creating tables and projections, 2017. `https://my.vertica.com/get-started-vertica/creating-tables-projections/`. Datum: 15. 4. 2017.

[17] HPE Vertica. Useful system tables, 2017. `https://my.vertica.com/get-started-vertica/useful-system-tables/`. Datum: 18. 4. 2017.

[18] Dennis Shasha and Philippe Bonnet. *Database Tuning: Principles, Experiments, and Troubleshooting Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. pp. 123-143. ISBN 978-1-55860-753-8.

[19] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., New York, NY, 6th edition, 2001. ISBN 0471417432.

[20] Synametrics Technologies. Top 10 performance tuning tips for relational databases, 2017. `https://synametrics.com/SynametricsWebApp/WPTop10Tips.jsp`. Datum: 22. 4. 2017.

[21] HPE Vertica. Understanding query plans, 2017. https://my.vertica.com/get-started-vertica/understanding-query-plans/. Datum: 23. 4. 2017.

[22] HPE Vertica. Reading query plans, 2017. https://my.vertica.com/kb/Reading-Query-Plans/Content/BestPractices/Reading-Query-Plans.htm. Datum: 12. 4. 2017.

[23] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill Higher Education, 7th edition, 2012. ISBN 978-0-07-338309-5.

[24] Transaction Processing Performance Council (TPC). TPC Benchmark^{TM} H – Standard Specification, 2017. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.1.pdf. Datum: 4. 4. 2017.

A Lock modes

Mode	Description
S – Shared	Use a Shared lock for SELECT queries that run at the serialized transaction isolation level. This allows queries to run concurrently, but the S lock creates the effect that transactions are running in serial order. The S lock ensures that one transaction does not affect another transaction until one transaction completes and its S lock is released.
I – Insert	Vertica requires an Insert lock to insert data into a table. Multiple transactions can lock an object in Insert mode simultaneously, enabling multiple inserts and bulk loads to occur at the same time. This behavior is critical for parallel loads and high ingestion rates.
SI – Shared Insert	Vertica requires a Shared Insert lock when both a read and an insert occur in a transaction. Shared Insert mode prohibits delete/update operations. An SI lock also results from lock promotion.
X – Exclusive	Vertica uses Exclusive locks when performing deletes and updates. Only mergeout and moveout operations (U locks) can run concurrently on objects with X locks.
T – Tuple Mover	The Tuple Mover uses T locks for operations on delete vectors. Tuple Mover operations upgrade the table lock mode from U to T when work on delete vectors starts so that no other updates or deletes can happen concurrently. T locks are also used for COPY into pre-join projections.
U – Usage	Vertica uses Usage locks for moveout and mergeout Tuple Mover operations. These Tuple Mover operations run automatically in the background, therefore, most other operations (except those requiring an O lock) can run when the object is locked in U mode.
O – Owner	An object acquires an Owner lock when it undergoes changes in both data and structure. Such changes can occur in some DDL operations, such as DROP_PARTITION, TRUNCATE TABLE, and ADD COLUMN.
IV – Insert-Validate	An Insert Validate lock is needed for insert operations where the system performs constraint validation for enabled PRIMARY or UNIQUE key constraints.

Table A.1: Lock modes that are available in Vertica. Descriptions are taken from the official documentation [8]. Terms *Tuple mover*, *mergeout operation* and *moveout operation* are defined in the documentation in chapter *Tuple Mover Operations*.

B Parameters of main function of Database Designer

Paramenter	**Description**
design_name	Name of the design that you want to populate and deploy, type VARCHAR.
output_design_file	Absolute path for saving the file that contains the DDL statements that create the design projections, type VARCHAR.
output_deployment_file	Absolute path for saving the file that contains the deployment script, type VARCHAR.
a *analyze_statistics*	(Optional) BOOLEAN that specifies whether or not to collect or refresh statistics for the tables before populating the design. Default is 'false'. Accurate statistics help Database Designer optimize compression and query performance. Updating statistics takes time and resources. If 'true', executes ANALYZE_STATISTICS. If ANALYZE_STATISTICS has run recently, set this parameter to 'false'.
deploy	(Optional) BOOLEAN that specifies whether or not to deploy the Database Designer design using the deployment script created by this function. Default: 'true'.
drop_design_workspace	(Optional) BOOLEAN that specifies whether or not to drop the design workspace after the design has been deployed. Default: 'true'.
continue_after_error	(Optional) BOOLEAN that specifies whether DE-SIGNER_RUN_POPULATE_DESIGN_AND_DEPLOY should continue running if an error occurs. Default: 'false'.

Table B.1: Parameters of function *DESIGNER_RUN_POPULATE_DE-SIGN_AND_DEPLOY*. Descriptions are taken from the official documentation [8].

C Output tables of DBPerfComp

C.1 Table – monitoring of query runs

```
CREATE TABLE monitoring_table (
    schema_name VARCHAR(30),
    start_timestamp TIMESTAMP,
    end_timestamp TIMESTAMP,
    transaction_id BIGINT,
    statement_id BIGINT,
    response_ms NUMERIC(20,5),
    memory_allocated_kb NUMERIC(20,5),
    memory_used_kb NUMERIC(20,5),
    cpu_time_ms BIGINT,
    label VARCHAR(100),
    query_name VARCHAR(100),
    runname VARCHAR(100),
    testname VARCHAR(100),
    user_name VARCHAR(30),
    resource_pool VARCHAR(30),
    request_id BIGINT)
```

C.2 Main table – processed and measured data

```
CREATE TABLE measurements (
    test_name VARCHAR(100),
    base_schema VARCHAR(100),
    compare_schema VARCHAR(100),
    max_set_depth Integer,
    query_set VARCHAR(100),
    query VARCHAR(100),
    design_time_ms FLOAT,
    ratio NUMERIC(20,5),
    ratio_query NUMERIC(20,5),
    sum_response_ms NUMERIC(20,5),
    avg_response_ms NUMERIC(20,5),
    max_response_ms NUMERIC(20,5),
    median_response_ms NUMERIC(20,5),
    sum_memory_allocated_kb NUMERIC(20,5),
    avg_memory_allocated_kb NUMERIC(20,5),
    max_memory_allocated_kb NUMERIC(20,5),
    median_memory_allocated_kb NUMERIC(20,5),
    sum_memory_used_kb NUMERIC(20,5),
    avg_memory_used_kb NUMERIC(20,5),
    max_memory_used_kb NUMERIC(20,5),
    median_memory_used_kb NUMERIC(20,5),
    sum_cpu_time_ms NUMERIC(20,5),
    avg_cpu_time_ms NUMERIC(20,5),
    max_cpu_time_ms NUMERIC(20,5),
    median_cpu_time_ms NUMERIC(20,5),
    row_count Integer,
    query_set_order1 VARCHAR(100),
    query_set_order2 VARCHAR(100),
    query_set_order3 VARCHAR(100),
    query_set_order4 VARCHAR(100),
    query_set_order5 VARCHAR(100),
    vertica_version varchar(50),
    node_count Integer,
    cpu_node_count Integer,
    ram_size_gb Decimal(15, 2),
    data_size_gb Decimal(15, 2),
    monitor_time TIMESTAMP)
```

C.3 Table – queries order in query workloads

```
CREATE TABLE queries_order_table (
    testname VARCHAR(100),
    query1 VARCHAR(100),
    query2 VARCHAR(100),
    query3 VARCHAR(100),
    query4 VARCHAR(100),
    query5 VARCHAR(100),
    query_set VARCHAR(100),
    processed BOOLEAN)
```

C.4 Logs

```
[TESTDESIGN - ADDING QUERY SET] Length of query bucket after adding: 680
[TESTDESIGN - ADDING QUERY SET] For query set 14,19 and for schema dbd_fk check
query 8 from the config file
[TESTDESIGN - ADDING QUERY SET] Query 8 from the config file is not in the query set
[TESTDESIGN - ADDING QUERY SET] In the query bucket there IS a duplicity for query
set 14,19,8
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET updated
[TESTDESIGN - ADDING QUERY SET] In the query bucket, the schema dbd_fk is missing
at the query set 14,19,8. »> Schema dbd_fk is addded. All schemas at this query set are:
dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 1. Schemas:
dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 10. Schemas:
dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 11. Schemas:
dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] For query set 14,19 and for
schema dbd_fk check query 8 from the config file
[TESTDESIGN - ADDING QUERY SET] Query 8 from the config file is not in the query set
[TESTDESIGN - ADDING QUERY SET] In the query bucket there IS a duplicity for query
set 14,19,8
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET updated
[TESTDESIGN - ADDING QUERY SET] In the query bucket, the schema
dbd_fk is missing at the query set 14,19,8. »> Schema dbd_fk is addded. All schemas at this
query set are: dbd_customized, dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 1.
Schemas: dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 10.
Schemas: dbd_customized,dbd_basic,dbd_fk
[TESTDESIGN - ADDING QUERY SET] QUERY BUCKET - Query set: 11.
Schemas: dbd_customized,dbd_basic,dbd_fk
```

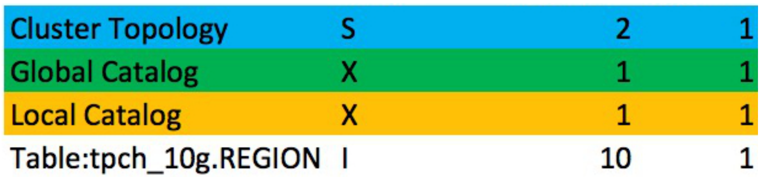
D Query for investigation of Database Designer influence on database objects

```
SELECT DISTINCT
    la.object_name la_object,
    la.mode,
    sum(timestampdiff('ms', la.time, lr.time)) total_lock_time_MS_sum,
    count(timestampdiff('ms', la.time, lr.time)) total_lock_time_MS_count,
    min(start_time),
    max(start_time)
FROM dc_lock_attempts la
    INNER JOIN dc_lock_releases lr ON la.transaction_id = lr.transaction_id
        AND la.session_id = lr.session_id AND la.node_name = lr.node_name
        AND la.object = lr.object AND la.object_name = lr.object_name
     INNER JOIN query_requests qr ON qr.transaction_id = la.transaction_id
WHERE
    request <> 'select 1'
    AND (qr.user_name = 'design_user' OR qr.request = '%testing_design%')
    AND start_time > '2017-01-01 00:00:00'
GROUP BY la.object_name, la.mode
ORDER BY la.object_name, la.mode
LIMIT 1000
```

E Locks – CRUD operations

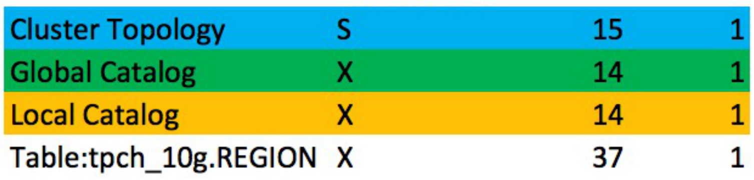
This appendix shows locks that were created during CRUD operations that worked with *Region* table. During *SELECT*, no lock was measured. Tables situated below present created locks during *INSERT*, *UPDATE* and *DELETE*. Clearly, except *Cluster Topology*, *Global Catalog*, and *Local Catalog*, only one object was influenced with a lock during any table modification.

INSERT



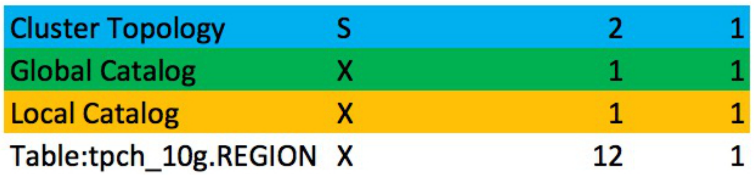
Cluster Topology	S	2	1
Global Catalog	X	1	1
Local Catalog	X	1	1
Table:tpch_10g.REGION	I	10	1

UPDATE



Cluster Topology	S	15	1
Global Catalog	X	14	1
Local Catalog	X	14	1
Table:tpch_10g.REGION	X	37	1

DELETE



Cluster Topology	S	2	1
Global Catalog	X	1	1
Local Catalog	X	1	1
Table:tpch_10g.REGION	X	12	1

F TPC-H queries created for the experiments

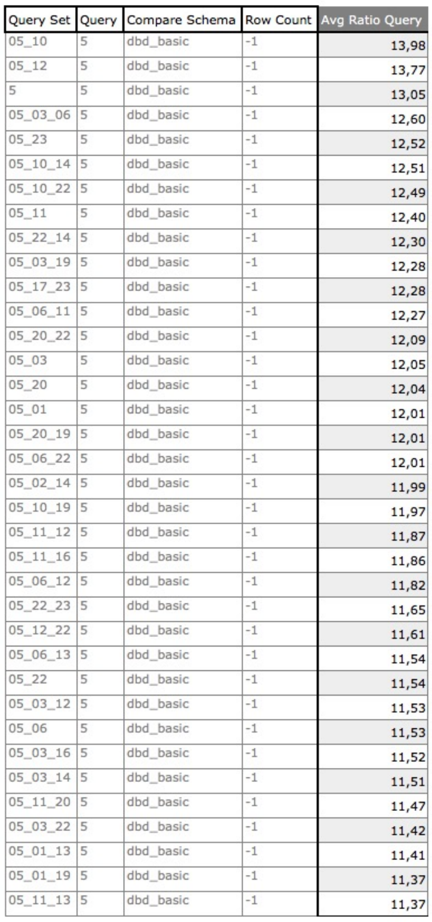
F.1 Query 23

```
SELECT  /*+ label(_LABEL_) */
    c_name,
    n_name,
    o_orderpriority,
    o_orderstatus,
    count(o_orderkey) as count_orders,
    cast(sum(o_totalprice) as DECIMAL(8,0)) as total_price,
    min(o_orderdate) as min_date,
    max(o_orderdate) as max_date,
    cast(sum(l_quantity) as DECIMAL(4,0)) as quantity
FROM
    customer,
    orders,
    lineitem,
    nation
WHERE
    o_custkey = c_custkey and
    l_orderkey = o_orderkey and
    o_orderpriority in ('1-URGENT', '2-HIGH') and
    n_regionkey = 3 and
    n_nationkey = c_nationkey
GROUP BY
    c_name,
    o_orderpriority,
    o_orderstatus,
    n_name
ORDER BY
    count_orders DESC, total_price DESC
LIMIT 100
```

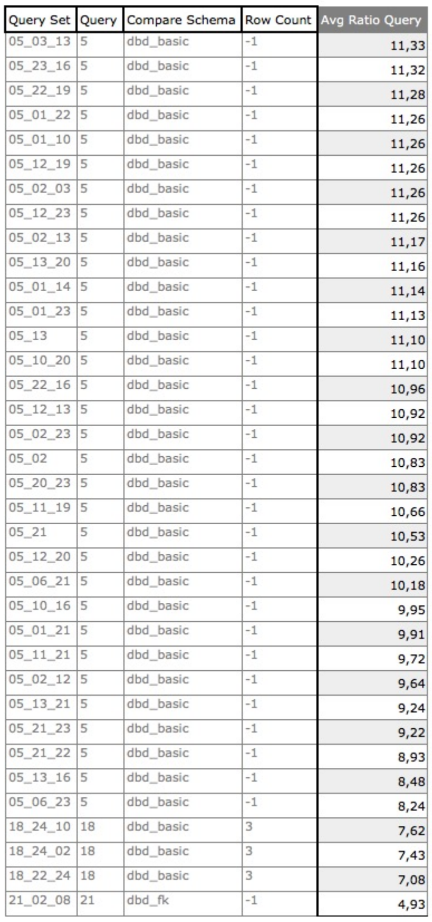
F.2 Query 24

```
SELECT  /*+ label(_LABEL_) */
    n_name,
    l_shipmode,
    c_mktsegment
FROM
    customer,
    orders,
    lineitem,
    nation,
    supplier
WHERE
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and n_nationkey = c_nationkey
    and s_suppkey = l_suppkey
    and o_orderdate >= date '1994-11-01'
    and c_acctbal > 9000
    and l_quantity > 2
    and n_name LIKE 'A%'
    and s_acctbal > 7000
    and l_shipmode in ('MAIL', 'REG AIR', 'AIR')
GROUP BY
    n_name,
    l_shipmode,
    c_mktsegment,
    o_orderpriority
```

G List of query workloads with timeout



Query Set	Query	Compare Schema	Row Count	Avg Ratio Query
05_10	5	dbd_basic	-1	13,98
05_12	5	dbd_basic	-1	13,77
5	5	dbd_basic	-1	13,05
05_03_06	5	dbd_basic	-1	12,60
05_23	5	dbd_basic	-1	12,52
05_10_14	5	dbd_basic	-1	12,51
05_10_22	5	dbd_basic	-1	12,49
05_11	5	dbd_basic	-1	12,40
05_22_14	5	dbd_basic	-1	12,30
05_03_19	5	dbd_basic	-1	12,28
05_17_23	5	dbd_basic	-1	12,28
05_06_11	5	dbd_basic	-1	12,27
05_20_22	5	dbd_basic	-1	12,09
05_03	5	dbd_basic	-1	12,05
05_20	5	dbd_basic	-1	12,04
05_01	5	dbd_basic	-1	12,01
05_20_19	5	dbd_basic	-1	12,01
05_06_22	5	dbd_basic	-1	12,01
05_02_14	5	dbd_basic	-1	11,99
05_10_19	5	dbd_basic	-1	11,97
05_11_12	5	dbd_basic	-1	11,87
05_11_16	5	dbd_basic	-1	11,86
05_06_12	5	dbd_basic	-1	11,82
05_22_23	5	dbd_basic	-1	11,65
05_12_22	5	dbd_basic	-1	11,61
05_06_13	5	dbd_basic	-1	11,54
05_22	5	dbd_basic	-1	11,54
05_03_12	5	dbd_basic	-1	11,53
05_06	5	dbd_basic	-1	11,53
05_03_16	5	dbd_basic	-1	11,52
05_03_14	5	dbd_basic	-1	11,51
05_11_20	5	dbd_basic	-1	11,47
05_03_22	5	dbd_basic	-1	11,42
05_01_13	5	dbd_basic	-1	11,41
05_01_19	5	dbd_basic	-1	11,37
05_11_13	5	dbd_basic	-1	11,37



Query Set	Query	Compare Schema	Row Count	Avg Ratio Query
05_03_13	5	dbd_basic	-1	11,33
05_23_16	5	dbd_basic	-1	11,32
05_22_19	5	dbd_basic	-1	11,28
05_01_22	5	dbd_basic	-1	11,26
05_01_10	5	dbd_basic	-1	11,26
05_12_19	5	dbd_basic	-1	11,26
05_02_03	5	dbd_basic	-1	11,26
05_12_23	5	dbd_basic	-1	11,26
05_02_13	5	dbd_basic	-1	11,17
05_13_20	5	dbd_basic	-1	11,16
05_01_14	5	dbd_basic	-1	11,14
05_01_23	5	dbd_basic	-1	11,13
05_13	5	dbd_basic	-1	11,10
05_10_20	5	dbd_basic	-1	11,10
05_22_16	5	dbd_basic	-1	10,96
05_12_13	5	dbd_basic	-1	10,92
05_02_23	5	dbd_basic	-1	10,92
05_02	5	dbd_basic	-1	10,83
05_20_23	5	dbd_basic	-1	10,83
05_11_19	5	dbd_basic	-1	10,66
05_21	5	dbd_basic	-1	10,53
05_12_20	5	dbd_basic	-1	10,26
05_06_21	5	dbd_basic	-1	10,18
05_10_16	5	dbd_basic	-1	9,95
05_01_21	5	dbd_basic	-1	9,91
05_11_21	5	dbd_basic	-1	9,72
05_02_12	5	dbd_basic	-1	9,64
05_13_21	5	dbd_basic	-1	9,24
05_21_23	5	dbd_basic	-1	9,22
05_21_22	5	dbd_basic	-1	8,93
05_13_16	5	dbd_basic	-1	8,48
05_06_23	5	dbd_basic	-1	8,24
18_24_10	18	dbd_basic	3	7,62
18_24_02	18	dbd_basic	3	7,43
18_22_24	18	dbd_basic	3	7,08
21_02_08	21	dbd_fk	-1	4,93

H Query workloads – ordered by ratio

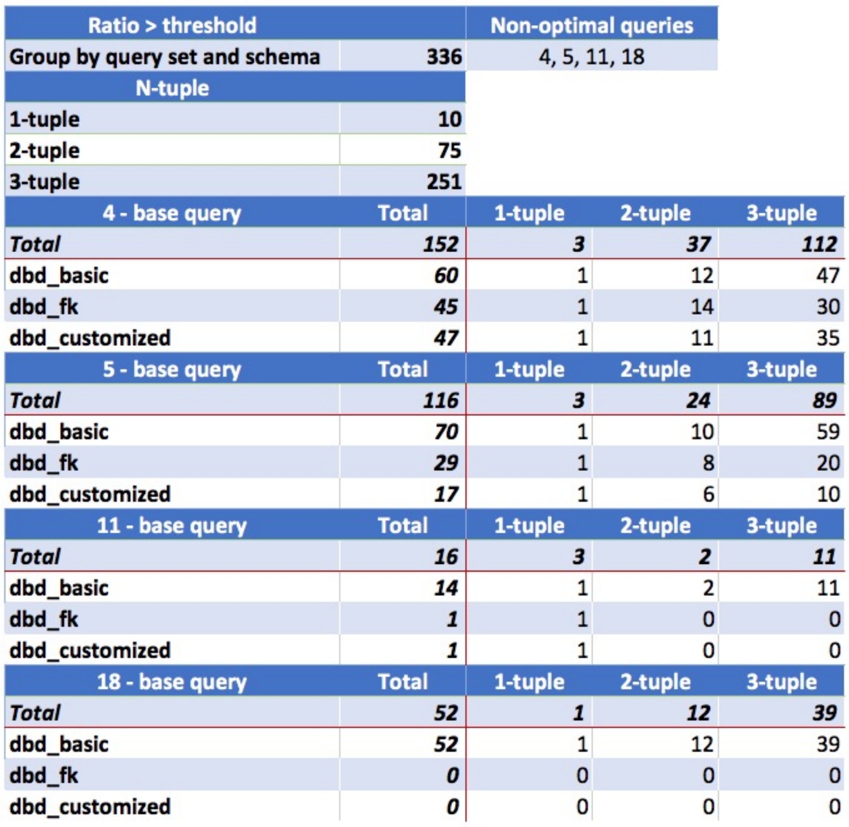
Query Set	Compare Schema	Avg Ratio
5	dbd_basic	13,05
05_23	dbd_basic	7,27
05_10	dbd_basic	7,11
05_12	dbd_basic	6,95
05_11	dbd_basic	6,53
05_03	dbd_basic	6,49
05_21	dbd_basic	6,41
05_01	dbd_basic	6,34
05_20	dbd_basic	6,33
05_22	dbd_basic	6,04
05_13	dbd_basic	5,88
05_06	dbd_basic	5,86
05_02	dbd_basic	5,74
05_22_23	dbd_basic	4,93
05_17_23	dbd_basic	4,92
05_02_14	dbd_basic	4,65
05_03_16	dbd_basic	4,65
05_03_06	dbd_basic	4,58
05_23_16	dbd_basic	4,58
05_03_19	dbd_basic	4,56
05_01_23	dbd_basic	4,53
05_02_03	dbd_basic	4,52
05_10_22	dbd_basic	4,47
05_02_13	dbd_basic	4,45
05_13_20	dbd_basic	4,43
05_12_23	dbd_basic	4,43
05_20_22	dbd_basic	4,42
05_11_16	dbd_basic	4,40
05_20_19	dbd_basic	4,40
05_20_23	dbd_basic	4,38
05_03_22	dbd_basic	4,38
05_22_14	dbd_basic	4,37
05_06_11	dbd_basic	4,35
05_10_14	dbd_basic	4,34
05_02_23	dbd_basic	4,33
05_21_23	dbd_basic	4,33
05_11_20	dbd_basic	4,30
05_03_13	dbd_basic	4,30
05_11_21	dbd_basic	4,26
05_03_14	dbd_basic	4,25
05_01_21	dbd_basic	4,24
05_06_21	dbd_basic	4,24
05_10_19	dbd_basic	4,23
05_11_12	dbd_basic	4,22
05_03_12	dbd_basic	4,20
05_06_22	dbd_basic	4,20
05_11_13	dbd_basic	4,20
05_01_19	dbd_basic	4,19
05_01_13	dbd_basic	4,17
05_06_13	dbd_basic	4,16
05_01_22	dbd_basic	4,15
05_22_16	dbd_basic	4,10
05_12_22	dbd_basic	4,09
05_01_10	dbd_basic	4,09
05_22_19	dbd_basic	4,09
05_11_19	dbd_basic	4,07
05_10_20	dbd_basic	4,06
05_06_12	dbd_basic	4,03
05_01_14	dbd_basic	4,02
05_13_21	dbd_basic	4,01
18_22_24	dbd_basic	3,97
05_12_19	dbd_basic	3,93
05_21_22	dbd_basic	3,91
05_12_13	dbd_basic	3,88
05_12_20	dbd_basic	3,76
05_10_16	dbd_basic	3,71
05_02_12	dbd_basic	3,44
05_13_16	dbd_basic	3,42
05_06_23	dbd_basic	3,41
18_24_02	dbd_basic	2,83
18_24_10	dbd_basic	2,74
21_02_08	dbd_fk	2,31
21	dbd_basic	2,16
21_23	dbd_basic	2,01
18_24	dbd_basic	1,98
21	dbd_customized	1,94
21_23	dbd_fk	1,93
21	dbd_fk	1,90
18_21	dbd_basic	1,90
18_06_21	dbd_basic	1,85
05_18_24	dbd_customized	1,83
18_21_02	dbd_basic	1,80
04_21	dbd_basic	1,80
21_02_07	dbd_basic	1,79
18_13_14	dbd_basic	1,79
21_08	dbd_basic	1,79
18_06	dbd_basic	1,77
04_18_21	dbd_basic	1,72
04_18_24	dbd_basic	1,66
05_21	dbd_fk	1,66
18_13_24	dbd_basic	1,66
21_24	dbd_basic	1,64
05_18_24	dbd_basic	1,64
21_24	dbd_fk	1,64
18_13_21	dbd_basic	1,64
21_08_23	dbd_basic	1,63
21_17_23	dbd_basic	1,63
05_06_18	dbd_customized	1,61
18_21_24	dbd_basic	1,61
18_03_06	dbd_basic	1,60
21_03	dbd_customized	1,60
04_21_23	dbd_basic	1,60
05_03_18	dbd_basic	1,58
04_21_22	dbd_basic	1,58
21_23	dbd_customized	1,57
05_02_17	dbd_fk	1,57
04_18	dbd_basic	1,56
05_18_21	dbd_basic	1,55
04_21_24	dbd_basic	1,55
21_24	dbd_customized	1,55
04_18_06	dbd_basic	1,55
21_01_23	dbd_basic	1,54
21_08	dbd_fk	1,53
04_21_08	dbd_basic	1,51
21_08_23	dbd_fk	1,50
21_23_24	dbd_basic	1,49
18_23	dbd_basic	1,49
04_18_23	dbd_basic	1,48
18_03_21	dbd_basic	1,48
04_23	dbd_basic	1,48
18_06_23	dbd_basic	1,47
05_21_23	dbd_fk	1,47
21_11_23	dbd_basic	1,46
21_03_23	dbd_basic	1,44
21_22	dbd_basic	1,43
21_02_23	dbd_basic	1,43
21_16	dbd_fk	1,43
04_07_21	dbd_basic	1,43
04_18_13	dbd_basic	1,42
05_21_07	dbd_fk	1,42
05_06_18	dbd_basic	1,42
04_21_16	dbd_basic	1,42
05_03_23	dbd_basic	1,42
18_07_21	dbd_basic	1,42
05_03_21	dbd_fk	1,41
21_07_08	dbd_basic	1,41
21_07	dbd_customized	1,41
21_07_23	dbd_basic	1,41
18_13	dbd_basic	1,40
18_01_21	dbd_basic	1,40
18_21_20	dbd_basic	1,40
04_21_01	dbd_basic	1,40
21_03	dbd_basic	1,40
21_03_24	dbd_basic	1,40
21_11	dbd_basic	1,39
21_16_23	dbd_basic	1,39
21_23_24	dbd_fk	1,39
21_08_24	dbd_basic	1,39
18_07_24	dbd_basic	1,39
21_16	dbd_basic	1,38
21_07_24	dbd_fk	1,38
18_23_24	dbd_basic	1,38
21_07	dbd_fk	1,38
21_03	dbd_fk	1,37
21_07	dbd_basic	1,37
05_20_21	dbd_basic	1,36
21_20_23	dbd_basic	1,36
21_13	dbd_basic	1,36
21_13_23	dbd_basic	1,36
21_07_23	dbd_fk	1,35
18_06_13	dbd_basic	1,35
21_17	dbd_customized	1,35
04_23_24	dbd_basic	1,35
04_21_20	dbd_basic	1,34
21_02	dbd_basic	1,34
04_03_21	dbd_basic	1,34
05_03_21	dbd_basic	1,34
18_01_19	dbd_basic	1,34
21_20	dbd_basic	1,34
05_21	dbd_customized	1,33
18_06_07	dbd_basic	1,33
21_03_08	dbd_basic	1,33
05_21_16	dbd_basic	1,33
21_01	dbd_basic	1,33
18_03_24	dbd_basic	1,33
05_08_21	dbd_basic	1,32
18_21_23	dbd_basic	1,32
21_03_23	dbd_fk	1,32
04_21_13	dbd_basic	1,32
05_21_24	dbd_fk	1,31
21_22	dbd_customized	1,31
04_18_16	dbd_basic	1,31
21_16_23	dbd_fk	1,31
21_13_23	dbd_fk	1,31
05_21_07	dbd_basic	1,31
21_08_22	dbd_basic	1,31
21_17	dbd_basic	1,31
21_20	dbd_fk	1,31
18_22_23	dbd_basic	1,30
05_08	dbd_basic	1,30
21_22	dbd_fk	1,30
04_23_20	dbd_basic	1,30
21_01_08	dbd_basic	1,30
21_07_22	dbd_basic	1,30
18_16_24	dbd_basic	1,29
18_06_24	dbd_basic	1,29
18_06_08	dbd_basic	1,29
21_02_22	dbd_basic	1,29

Query Set	Compare Schema	Avg Ratio
04_21_10	dbd_basic	1,29
04_23_19	dbd_basic	1,29
04_07_18	dbd_basic	1,29
05_08_21	dbd_fk	1,29
21_13	dbd_fk	1,29
04_21_02	dbd_basic	1,29
18_24_17	dbd_basic	1,29
18_16_21	dbd_basic	1,28
05_21_16	dbd_fk	1,28
04_03_18	dbd_basic	1,28
18_16_12	dbd_basic	1,28
21_03_07	dbd_basic	1,28
21_01_23	dbd_fk	1,28
04_18_08	dbd_basic	1,28
21_11_23	dbd_fk	1,28
21_14	dbd_basic	1,27
05_21_24	dbd_basic	1,27
21_03_16	dbd_basic	1,27
04_21_19	dbd_basic	1,27
21_22_23	dbd_basic	1,27
21_07_24	dbd_basic	1,27
21_08_16	dbd_basic	1,27
05_08_23	dbd_fk	1,27
05_21_07	dbd_customized	1,27
21_02_08	dbd_basic	1,27
21_08_11	dbd_basic	1,27
04_21_11	dbd_basic	1,27
21_22_23	dbd_fk	1,27
18_21_19	dbd_basic	1,26
21_07_22	dbd_fk	1,26
21_17	dbd_fk	1,26
18_06_16	dbd_basic	1,26
21_08_20	dbd_basic	1,26
18_13_16	dbd_basic	1,25
18_07_13	dbd_basic	1,25
21_20_23	dbd_fk	1,25
05_20_21	dbd_fk	1,25
21_14_23	dbd_basic	1,25
05_21_19	dbd_basic	1,25
05_18	dbd_fk	1,25
4	dbd_basic	1,24
21_11_24	dbd_fk	1,24
18_07_23	dbd_basic	1,24
21_23_24	dbd_customized	1,24
05_01_21	dbd_fk	1,24
18_08_21	dbd_basic	1,23
18_03	dbd_basic	1,23
05_03_21	dbd_customized	1,23
23	dbd_fk	1,23
21_17_24	dbd_fk	1,23
21_13_24	dbd_basic	1,23
05_21_22	dbd_fk	1,23
21_13_23	dbd_customized	1,23
04_23_11	dbd_basic	1,23
21_06	dbd_basic	1,23
05_21_22	dbd_customized	1,23
05_13_21	dbd_fk	1,22
21_16_24	dbd_basic	1,22
05_18_07	dbd_basic	1,22
21_19_23	dbd_basic	1,22
18_24_19	dbd_basic	1,22
18_21_22	dbd_basic	1,22
18	dbd_basic	1,22
21_03_08	dbd_fk	1,22
21_20_24	dbd_fk	1,21
21_08	dbd_customized	1,21
18_01_06	dbd_basic	1,21

Query Set	Compare Schema	Avg Ratio
21_07_24	dbd_customized	1,21
21_08_13	dbd_basic	1,21
18_16	dbd_basic	1,21
21_19	dbd_basic	1,20
21_08_22	dbd_customized	1,20
04_03_22	dbd_basic	1,20
21_02_03	dbd_fk	1,20
21_07_08	dbd_fk	1,20
21_03_07	dbd_fk	1,20
21_06_23	dbd_basic	1,19
21_01	dbd_fk	1,19
04_21_12	dbd_basic	1,19
18_21_14	dbd_basic	1,19
21_06_08	dbd_basic	1,19
21_22_24	dbd_basic	1,19
21_11	dbd_fk	1,19
21_08_17	dbd_basic	1,19
21_10_23	dbd_basic	1,19
05_02_21	dbd_basic	1,19
21_07_16	dbd_basic	1,19
21_22_24	dbd_customized	1,19
21_03_07	dbd_customized	1,19
18_03_13	dbd_basic	1,18
04_18_01	dbd_basic	1,18
05_21_24	dbd_customized	1,18
04_07	dbd_basic	1,18
18_22	dbd_basic	1,18
21_08_22	dbd_fk	1,18
05_18_23	dbd_basic	1,18
05_17	dbd_fk	1,18
21_08_19	dbd_basic	1,18
05_13_18	dbd_basic	1,18
21_06	dbd_fk	1,17
18_21_12	dbd_basic	1,17
05_21_16	dbd_customized	1,17
21_01_03	dbd_basic	1,17
5	dbd_customized	1,17
05_21_14	dbd_basic	1,17
21_08_24	dbd_fk	1,17
21_03_24	dbd_fk	1,17
21_01_11	dbd_basic	1,16
21_16_24	dbd_fk	1,16
04_21_17	dbd_basic	1,16
21_07_13	dbd_fk	1,16
04_23_16	dbd_basic	1,16
05_18	dbd_basic	1,16
21_22_24	dbd_fk	1,16
05_11_21	dbd_fk	1,16
05_21_19	dbd_fk	1,16
21_08_10	dbd_basic	1,16
05_21_23	dbd_customized	1,16
21_11_24	dbd_basic	1,16
23_03	dbd_fk	1,16
18_03_23	dbd_basic	1,16
21_01_13	dbd_basic	1,16
18_13_19	dbd_basic	1,15
21_23_12	dbd_basic	1,15
18_21_10	dbd_basic	1,15
18_16_23	dbd_basic	1,15
21_16	dbd_customized	1,15
04_23_08	dbd_basic	1,15
21_03_13	dbd_basic	1,15
04_18_22	dbd_basic	1,15
18_21_11	dbd_basic	1,15
05_17_18	dbd_basic	1,15
18_21_17	dbd_basic	1,15
21_01_16	dbd_basic	1,15

Query Set	Compare Schema	Avg Ratio
21_02_20	dbd_basic	1,14
21_01_03	dbd_fk	1,14
18_01_13	dbd_basic	1,14
21_08_14	dbd_basic	1,14
21_13_24	dbd_customized	1,14
21_02_23	dbd_fk	1,14
04_03_23	dbd_basic	1,14
21_06_24	dbd_basic	1,14
04_03_08	dbd_basic	1,14
18_07	dbd_basic	1,14
04_18_11	dbd_basic	1,14
05_10_21	dbd_fk	1,14
21_20_24	dbd_basic	1,14
23_08	dbd_fk	1,14
18_06_22	dbd_basic	1,14
21_07_17	dbd_customized	1,14
18_13_20	dbd_basic	1,14
21_02_03	dbd_basic	1,13
21_22_23	dbd_customized	1,13
18_08_22	dbd_basic	1,13
21_17_22	dbd_basic	1,13
21_02_24	dbd_basic	1,13
21_03_17	dbd_customized	1,13
5	dbd_fk	1,13
21_08_19	dbd_fk	1,13
05_18_21	dbd_customized	1,13
21_01_20	dbd_basic	1,13
21_13_24	dbd_fk	1,12
21_19	dbd_fk	1,12
05_08	dbd_fk	1,12
21_07_23	dbd_customized	1,12
21_13_20	dbd_fk	1,12
04_23_13	dbd_basic	1,12
05_20_14	dbd_basic	1,12
21_10	dbd_basic	1,12
21_20	dbd_customized	1,12
21_13_16	dbd_basic	1,12
18_06_20	dbd_basic	1,12
18_01	dbd_basic	1,12
04_23_01	dbd_basic	1,11
04_03	dbd_basic	1,11
05_03_08	dbd_basic	1,11
21_02_11	dbd_basic	1,11
21_07_13	dbd_basic	1,11
21_10	dbd_fk	1,11
21_02	dbd_fk	1,11
21_08_11	dbd_fk	1,11
21_11	dbd_customized	1,11
21_03_17	dbd_fk	1,11
18_08_13	dbd_basic	1,11
21_01_20	dbd_fk	1,11
18_08	dbd_basic	1,11
21_01_07	dbd_fk	1,11
21_03_13	dbd_customized	1,11
21_13_20	dbd_basic	1,10
05_17_21	dbd_basic	1,10
18_01_23	dbd_basic	1,10
21_16_23	dbd_customized	1,10
18_06_12	dbd_basic	1,10
21_13	dbd_customized	1,10
18_13_11	dbd_basic	1,10
21_01_19	dbd_basic	1,10
21_03_11	dbd_basic	1,10
05_12_21	dbd_fk	1,10
05_02_21	dbd_fk	1,10
21_03_17	dbd_basic	1,10
21_07_08	dbd_customized	1,10

I Cluster2 – results



Ratio > threshold		Non-optimal queries		
Group by query set and schema	336	4, 5, 11, 18		
N-tuple				
1-tuple	10			
2-tuple	75			
3-tuple	251			
4 - base query	Total	1-tuple	2-tuple	3-tuple
Total	*152*	*3*	*37*	*112*
dbd_basic	*60*	1	12	47
dbd_fk	*45*	1	14	30
dbd_customized	*47*	1	11	35
5 - base query	Total	1-tuple	2-tuple	3-tuple
Total	*116*	*3*	*24*	*89*
dbd_basic	*70*	1	10	59
dbd_fk	*29*	1	8	20
dbd_customized	*17*	1	6	10
11 - base query	Total	1-tuple	2-tuple	3-tuple
Total	*16*	*3*	*2*	*11*
dbd_basic	*14*	1	2	11
dbd_fk	*1*	1	0	0
dbd_customized	*1*	1	0	0
18 - base query	Total	1-tuple	2-tuple	3-tuple
Total	*52*	*1*	*12*	*39*
dbd_basic	*52*	1	12	39
dbd_fk	*0*	0	0	0
dbd_customized	*0*	0	0	0

Table I.1: Distribution of non-optimal *query-schema sets* among particular *base query workloads – cluster2*.

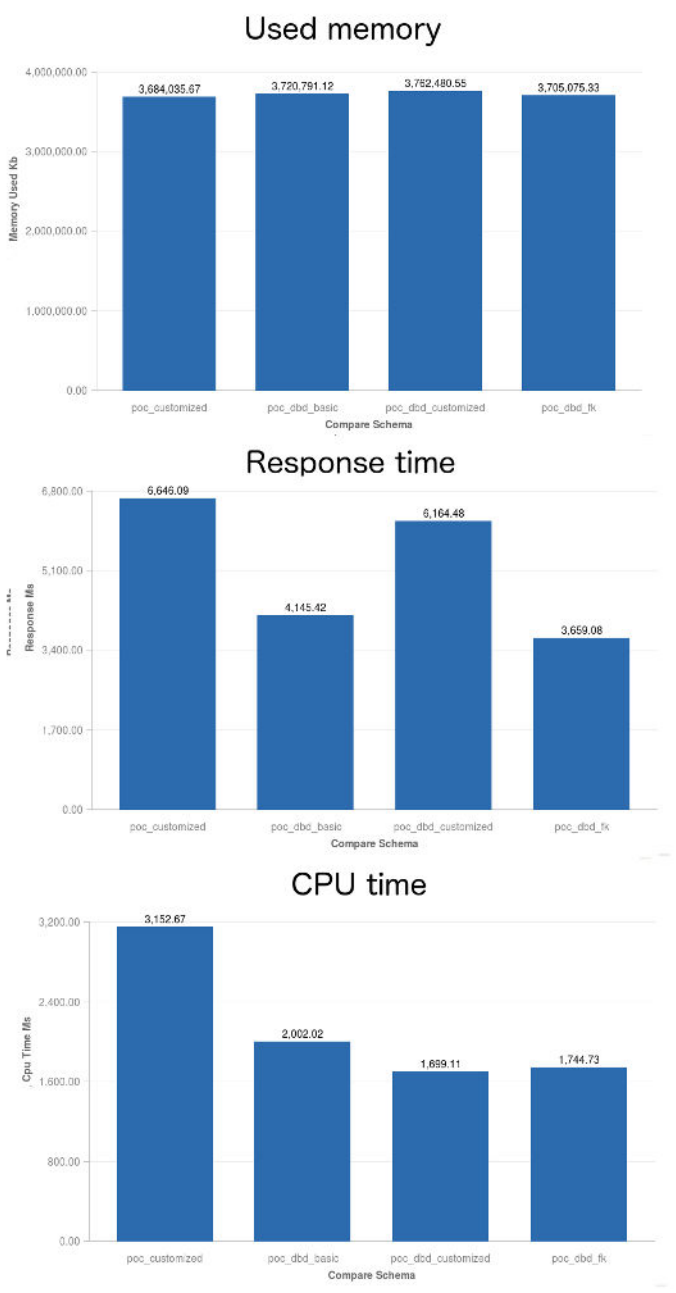


Figure I.1: Graphs of allocated memory, used memory, CPU time and response time where only singletons are selected – *cluster2*.

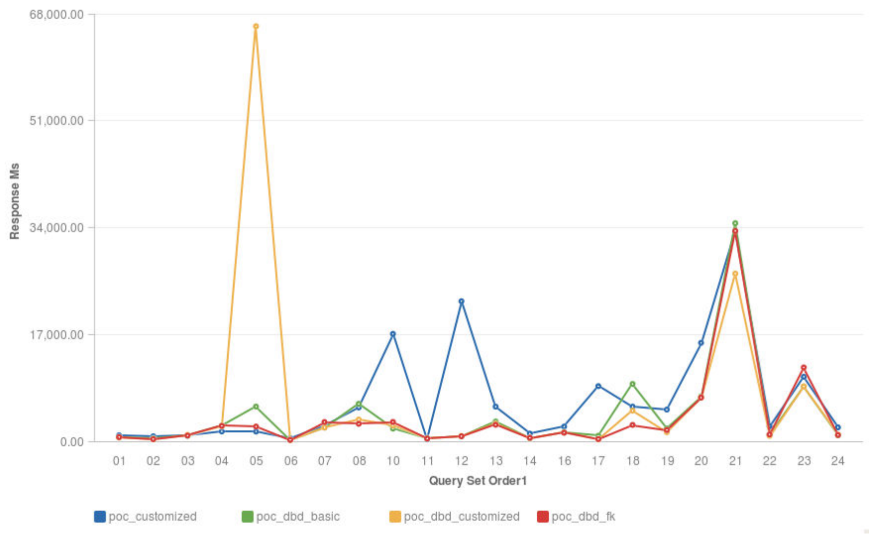


Figure I.2: Response time for particular singletons and schemata – *cluster2*.

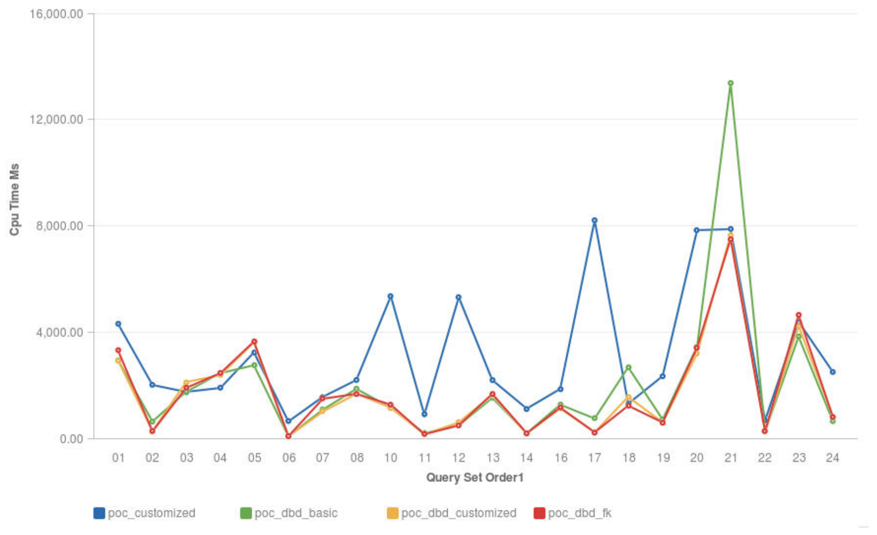


Figure I.3: CPU time for particular singletons and schemata – *cluster2*.

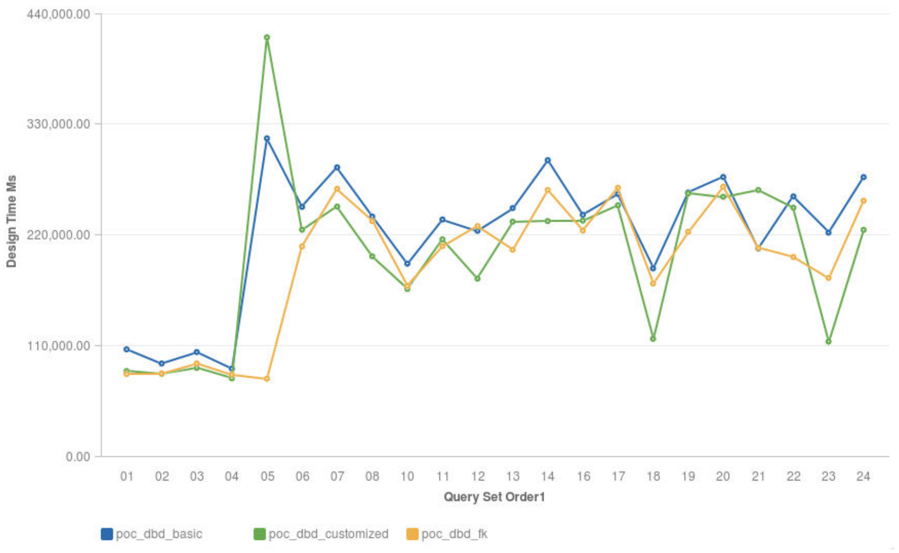


Figure I.4: Design time for particular singletons and schemata – *cluster2*.

J Query for investigation of a query plan

```
SELECT
  x.path_id,
  epp.path_line_index,
  to_char(x.db_time, :fmt12) as db_time, to_char(x.cpu_time, :fmt12) as cpu_time,
  to_char(x.read_bytes_cache, :fmt12) as read_bytes_cache, to_char(x.read_bytes_disk,
        :fmt12) as read_bytes_disk,
  to_char(x.memory_allocated, :fmt12) as memory_allocated, to_char(rows_produced,
        :fmt12) as rows_produced,
  to_char(x.wos_bytes, :fmt12) as wos_bytes, to_char(ros_bytes, :fmt12) as ros_bytes,
  to_char(x.temp_bytes_raw, :fmt12) as temp_bytes_raw, to_char(x.temp_bytes, :fmt12)
        as temp_bytes,
  to_char(x.queue_wait_time, :fmt12) as queue_wait_time, to_char(x.network_wait_time,
        :fmt12) as network_wait_time,
  to_char(x.merge_phases, :fmt12) as merge_phases,
  x.statement_id,
  epp.path_line
FROM (
   SELECT
     ep.node_name, ep.transaction_id, ep.statement_id,
     ep.path_id,
     ep.path_line,
     round(sum(decode(zz.counter_name, 'clock time (us)', zz.counter_value, 0))/1000)
           as db_time,
     round(sum(decode(zz.counter_name, 'execution time (us)', zz.counter_value, 0))/1000)
           as cpu_time,
     round(sum(decode(zz.counter_name, 'input queue wait (us)', zz.counter_value, 0))/1000)
           as queue_wait_time,
     round(sum(decode(zz.counter_name, 'network wait (us)', zz.counter_value, 0))/1000)
           as network_wait_time,
     sum(decode(zz.counter_name, 'bytes read from cache', zz.counter_value, 0))
           as read_bytes_cache,
     sum(decode(zz.counter_name, 'bytes read from disk', zz.counter_value, 0))
           as read_bytes_disk,
     sum(decode(zz.counter_name, 'cumulative size of raw temp data (bytes)',
           zz.counter_value, 0)) as temp_bytes_raw,
     sum(decode(zz.counter_name, 'cumulative size of temp files (bytes)', zz.counter_value,
           0)) as temp_bytes,
     sum(decode(zz.counter_name, 'WOS bytes written', zz.counter_value, 0)) as wos_bytes,
     sum(decode(zz.counter_name, 'ROS bytes written', zz.counter_value, 0)) as ros_bytes,
     sum(decode(zz.counter_name, 'completed merge phases', zz.counter_value, 0))
           as merge_phases,
     sum(decode(zz.counter_name, 'memory allocated (bytes)', zz.counter_value, 0))
           as memory_allocated,
     sum(decode(zz.counter_name, 'rows produced', zz.counter_value, 0)) as rows_produced
   FROM dc_explain_plans ep
   JOIN dc_requests_issued ri
     USING (transaction_id, statement_id)
   LEFT OUTER JOIN execution_engine_profiles zz
     USING (transaction_id, statement_id, path_id)
   WHERE
     ri.label = '_LABEL_'
     AND ep.path_line_index = 1
   GROUP BY
     ep.node_name, ep.transaction_id, ep.statement_id,
     ep.path_id,
     ep.path_line
) x
JOIN dc_explain_plans epp
  USING (transaction_id, statement_id, path_id)
ORDER BY
  epp.time
```

K Simplified Query 5

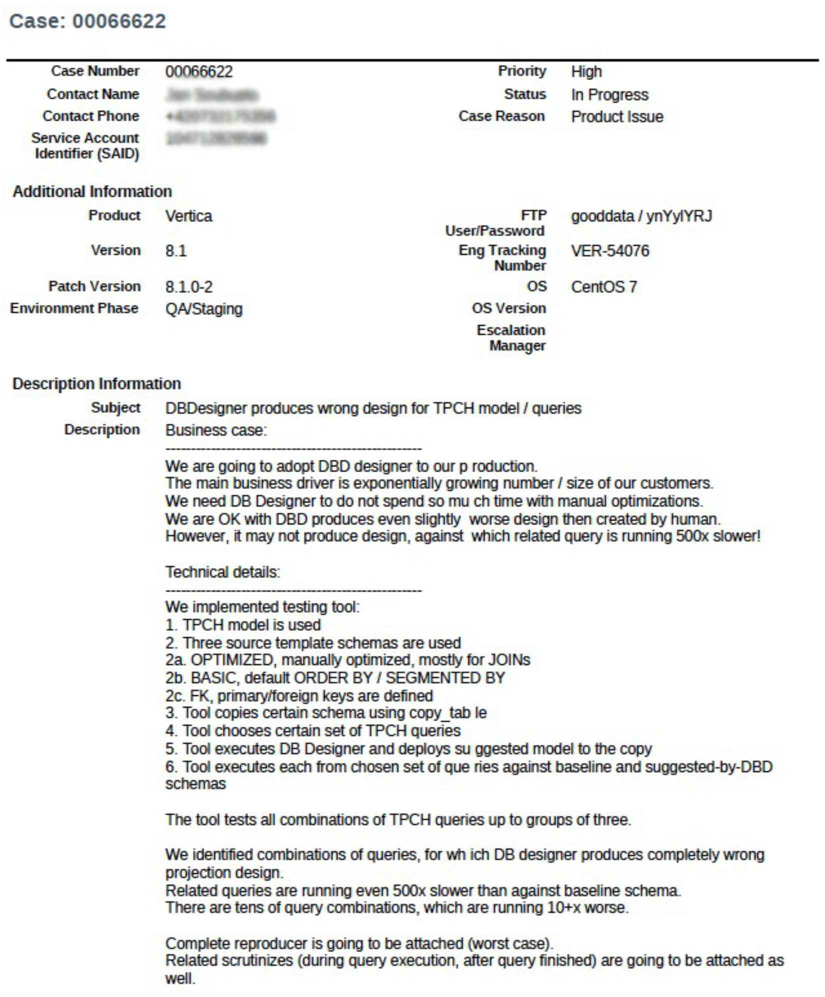
```
SELECT
    s_name,
    o_orderkey,
    l_linestatus
FROM
   customer,
   orders,
   lineitem,
   supplier
WHERE
   c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and l_suppkey = s_suppkey
   and c_nationkey = s_nationkey
   and o_orderdate >= date ’1997-01-01’
   and o_orderdate < date ’1997-01-01’ + interval ’1’ year
ORDER BY
   l_extendedprice * (1 - l_discount) desc
LIMIT 1
```

L Query for investigation of query operations

```
SELECT
  ri.label,
  operator_name, counter_name,
  (sum(counter_value)/1000)::integer as duration_ms,
  count(*) as threads,
  (sum(counter_value)/1000/count(*))::integer as duration_ms_per_thread
FROM
  execution_engine_profiles_snap ee
JOIN (
  SELECT distinct transaction_id, statement_id, label
  FROM dc_requests_issued_snap_hist
  WHERE regexp_like(label, '^_LABEL_$')
) ri
  USING (transaction_id, statement_id)
WHERE
  counter_name like '\%(us)'
GROUP BY
  ri.label, operator_name, counter_name
HAVING
  (sum(counter_value)/1000/count(*))::integer > 5
ORDER BY
  duration_ms desc
LIMIT 30
```

M Bug report

This appendix includes official bug report created by GoodData s.r.o. and sent to Vertica. Selected emails from the conversation between Vertica and database administrator of GoodData s.r.o. are included.



Case: 00066622

Case Number	00066622	Priority	High
Contact Name	[redacted]	Status	In Progress
Contact Phone	[redacted]	Case Reason	Product Issue
Service Account Identifier (SAID)	[redacted]		

Additional Information

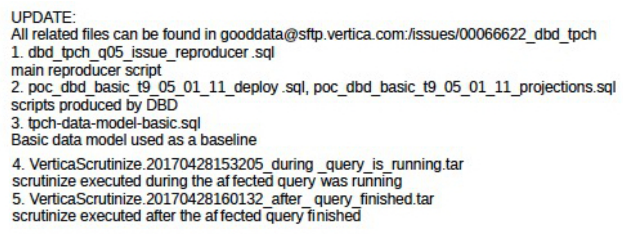
Product	Vertica	FTP User/Password	gooddata / ynYylYRJ
Version	8.1	Eng Tracking Number	VER-54076
Patch Version	8.1.0-2	OS	CentOS 7
Environment Phase	QA/Staging	OS Version	
		Escalation Manager	

Description Information

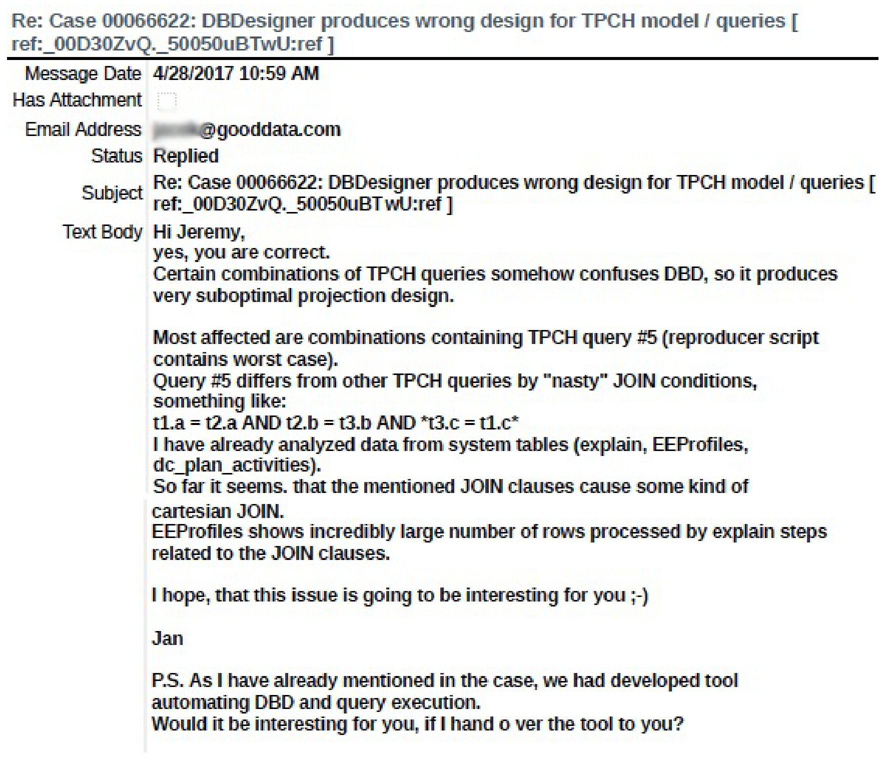
Subject: DBDesigner produces wrong design for TPCH model / queries

Description: Business case:

```
----------------------------------------------------
We are going to adopt DBD designer to our p roduction.
The main business driver is exponentially growing number / size of our customers.
We need DB Designer to do not spend so mu ch time with manual optimizations.
We are OK with DBD produces even slightly  worse design then created by human.
However, it may not produce design, against  which related query is running 500x slower!

Technical details:
----------------------------------------------------
We implemented testing tool:
1. TPCH model is used
2. Three source template schemas are used
2a. OPTIMIZED, manually optimized, mostly for JOINs
2b. BASIC, default ORDER BY / SEGMENTED BY
2c. FK, primary/foreign keys are defined
3. Tool copies certain schema using copy_tab le
4. Tool chooses certain set of TPCH queries
5. Tool executes DB Designer and deploys su ggested model to the copy
6. Tool executes each from chosen set of que ries against baseline and suggested-by-DBD
schemas

The tool tests all combinations of TPCH queries up to groups of three.

We identified combinations of queries, for wh ich DB designer produces completely wrong
projection design.
Related queries are running even 500x slower than against baseline schema.
There are tens of query combinations, which are running 10+x worse.

Complete reproducer is going to be attached (worst case).
Related scrutinizes (during query execution, after query finished) are going to be attached as
well.
```



UPDATE:
All related files can be found in gooddata@sftp.vertica.com:/issues/00066622_dbd_tpch
1. dbd_tpch_q05_issue_reproducer.sql
main reproducer script
2. poc_dbd_basic_t9_05_01_11_deploy.sql, poc_dbd_basic_t9_05_01_11_projections.sql
scripts produced by DBD
3. tpch-data-model-basic.sql
Basic data model used as a baseline
4. VerticaScrutinize.20170428153205_during _query_is_running.tar
scrutinize executed during the af fected query was running
5. VerticaScrutinize.20170428160132_after_ query_finished.tar
scrutinize executed after the af fected query finished



Re: Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	4/28/2017 10:59 AM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Replied
Subject	Re: Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Hi Jeremy,

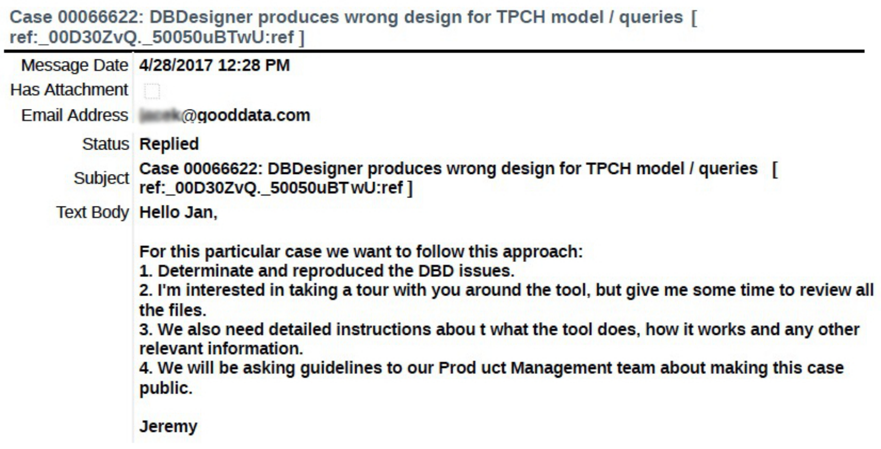
yes, you are correct.
Certain combinations of TPCH queries somehow confuses DBD, so it produces very suboptimal projection design.

Most affected are combinations containing TPCH query #5 (reproducer script contains worst case).
Query #5 differs from other TPCH queries by "nasty" JOIN conditions, something like:
t1.a = t2.a AND t2.b = t3.b AND *t3.c = t1.c*
I have already analyzed data from system tables (explain, EEProfiles, dc_plan_activities).
So far it seems. that the mentioned JOIN clauses cause some kind of cartesian JOIN.
EEProfiles shows incredibly large number of rows processed by explain steps related to the JOIN clauses.

I hope, that this issue is going to be interesting for you ;-)

Jan

P.S. As I have already mentioned in the case, we had developed tool automating DBD and query execution.
Would it be interesting for you, if I hand o ver the tool to you?

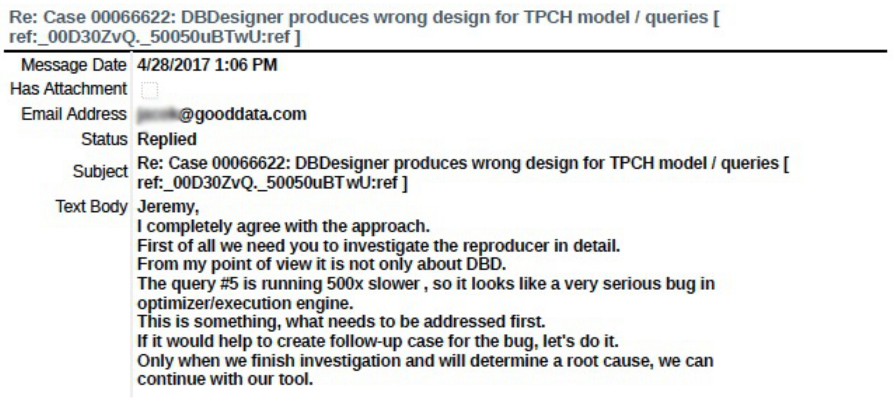


Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	4/28/2017 12:28 PM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Replied
Subject	Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Hello Jan,

For this particular case we want to follow this approach:
1. Determinate and reproduced the DBD issues.
2. I'm interested in taking a tour with you around the tool, but give me some time to review all the files.
3. We also need detailed instructions abou t what the tool does, how it works and any other relevant information.
4. We will be asking guidelines to our Prod uct Management team about making this case public.

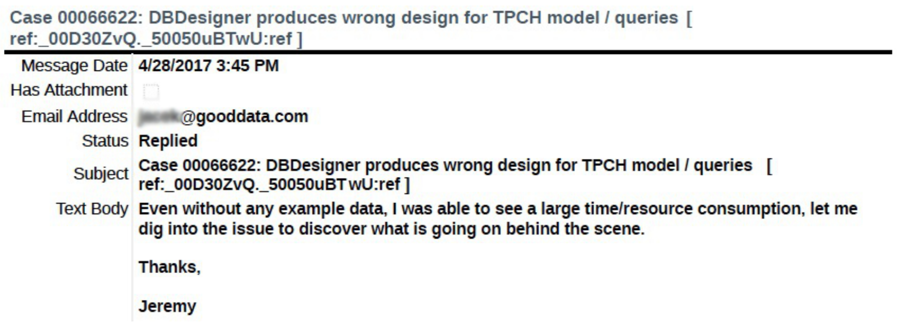
Jeremy



Re: Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	4/28/2017 1:06 PM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Replied
Subject	Re: Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Jeremy,

I completely agree with the approach.
First of all we need you to investigate the reproducer in detail.
From my point of view it is not only about DBD.
The query #5 is running 500x slower , so it looks like a very serious bug in optimizer/execution engine.
This is something, what needs to be addressed first.
If it would help to create follow-up case for the bug, let's do it.
Only when we finish investigation and will determine a root cause, we can continue with our tool.

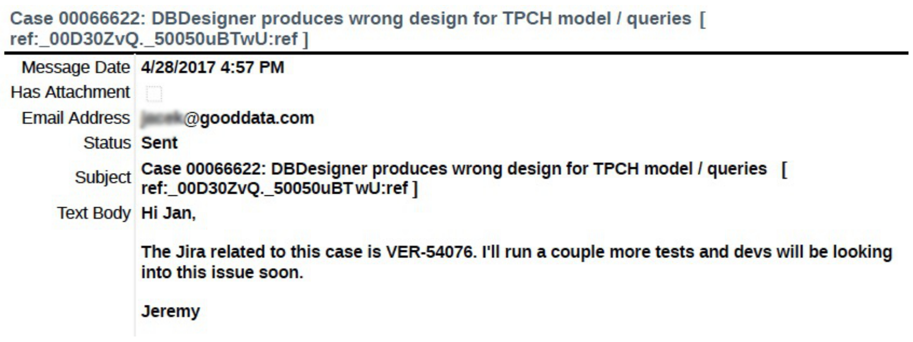


Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	4/28/2017 3:45 PM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Replied
Subject	Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Even without any example data, I was able to see a large time/resource consumption, let me dig into the issue to discover what is going on behind the scene.

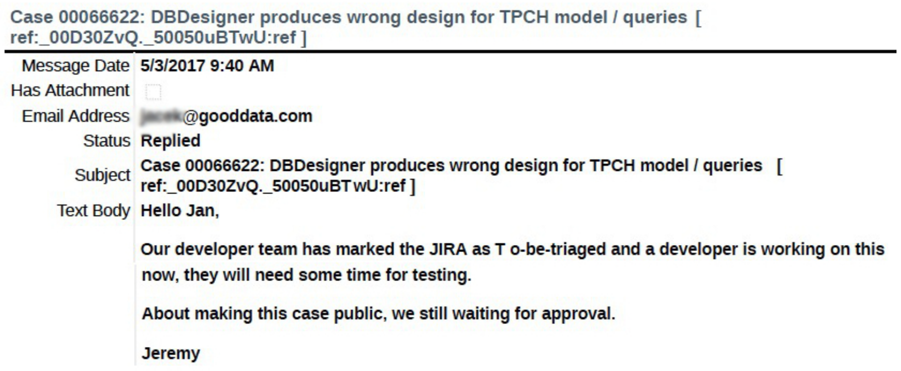
Thanks,

Jeremy



Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	4/28/2017 4:57 PM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Sent
Subject	Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Hi Jan, The Jira related to this case is VER-54076. I'll run a couple more tests and devs will be looking into this issue soon. Jeremy



Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]

Message Date	5/3/2017 9:40 AM
Has Attachment	☐
Email Address	[redacted]@gooddata.com
Status	Replied
Subject	Case 00066622: DBDesigner produces wrong design for TPCH model / queries [ref:_00D30ZvQ._50050uBTwU:ref]
Text Body	Hello Jan, Our developer team has marked the JIRA as T o-be-triaged and a developer is working on this now, they will need some time for testing. About making this case public, we still waiting for approval. Jeremy