VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SAP HANA PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL UHLÍŘ
AUTHOR

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PLATFORMA SAP HANA

SAP HANA PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
MICHAL UHLÍŘ

VEDOUCÍ PRÁCE
SUPERVISOR
Doc. Ing. JAROSLAV ZENDULKA, CSc.

BRNO 2014

Abstrakt

Tato práce pojednává o databázi pracující v paměti nazývané SAP HANA. Detailně popisuje architekturu a nové technologie, které tato databáze využívá. V další části se zabývá porovnáním rychlosti provedení vkládání a vybírání záznamů z databáze se stávající používanou relační databází MaxDB. Pro účely tohoto testování jsem vytvořil jednoduchou aplikaci v jazyce ABAP, která umožňuje testy provádět a zobrazuje jejich výsledky. Ty jsou shrnuty v poslední kapitole a ukazují SAP HANA jako jednoznačně rychlejší ve vybírání dat, avšak srovnatelnou, či pomalejší při vkládání dat do databáze. Přínos mé práce vidím v shrnutí podstatných změn, které s sebou data uložená v paměti přináší a názorné srovnání rychlosti provedení základních typů dotazů.

Abstract

This thesis discusses the in-memory database called SAP HANA. It describes in detail the architecture and new technologies used in this type of database. The next section presents a comparison of speed of the inserting and selecting data from the database with existing relational database MaxDB. For the purposes of this testing I created a simple application in ABAP language, which allows user to perform and display their results. These are summarized in the last chapter and demonstrate SAP HANA as clearly faster during selection of data, but comparable, or slower when inserting data into the database. I see contribution of my work in the summary of significant changes that come with data stored in the main memory and brings comparison of speed of basic types of queries.

Klíčová slova

SAP, SAP Hana, ABAP, Databáze pracující v paměti, Výkonostní testování, Slovníkové kódování

Keywords

SAP, SAP Hana, ABAP, In-Memory Database, Performance testing, Dictionary encoding

Citation

Michal Uhlíř: SAP HANA Platform, bakalářská práce, Brno, FIT VUT v Brně, 2014

SAP HANA Platform

Declaration

I declare that this thesis in my own work that has been created under the supervision of Doc. Ing. Jaroslav Zendulka, CSc., and all sources and literature that I have used during elaboration of the thesis are correctly cited with complete reference to the corresponding sources.

....
Michal Uhlíř
May 21, 2014

Acknowledgements

I would like to thank to my supervisor Doc. Ing. Jaroslav Zendulka, CSc., who supported me throughout my work on this thesis, you have been a tremendous mentor for me. I would especially like to thank to employees of SAP Brno for their time spent with me discussing SAP Hana and ABAP development. Furthermore I would also like to thank my parents and girlfiend for their endless love and support during my study.

Contents

List of Figures

Chapter 1

Introduction

The topic of my thesis is SAP HANA platform, a revolutionary approach to databases, developed by SAP under the leadership of prof. Hasso Plattner. This platform is based on a database running in the main memory, promising users on the fly processing of their data and a single data source. However, it also brings a lot of new options for the treatment of data in the database.

Even while studying faculty of information technologies in Brno, I was very interested in databases introduced in Database Systems course. In the second year, I got a unique chance to work as a tester in the SAP company, branch Brno. The most commonly used word in internal newsletters, or during the conversations with colleagues was HANA. So I started with research, what does HANA mean and what so revolutionary is hiding behind this word. Combined with my interest in the database, I presented my plan to prof. Zendulka, who offered me his cooperation very willingly.

The main goal of my work is to present the current situation in the processing of data from the database and the possibility of change that brings rapid development of technology. The following is a description of the architecture of in-memory database and technological changes needed for this architecture. One of the advantages is to store data in a columnar layout and the ability to encode data using the dictionary encoding. My job was also to create a test application that compares the performance of MaxDB, the underlying relational database and newly promoted SAP HANA database. This application is written in ABAP and allows user to export test results into the prepared Excel file. These results are described in detail in last chapter.

Chapter 2

Motivation

Computer science is reaching a new age. Companies need to preserve more data in database and to perform real time analytic operations with them. Very popular is storing unstructured data in database or just using tablets and smart phones to operate with our information system on a business trip, customer meeting or in public transport. In this chapter I would like to introduce enterprise computing, talk about new requirements for enterprise computing, analyze todays data management systems and describe changes of modern hardware performance.

2.1 Enterprise Computing

Enterprise computing is sold to business corporations as a complex solution of their corporate governance. Entire platform can be applied as packages within each area. These are accounting, business intelligence, human resource management, content management system or customer relationship management. Customers can choose from those packages and customize them for their use. That means, the entire company uses one information system with their own or rent database to store data.

2.2 Current Approach

Nowadays in enterprise computing exist two approaches. Each company has two databases. First one is optimized for inserting data and servicing many transactions at one time. Data is always actual and a lot of users can access them. Second one is created by regular copying of the first database and optimized for analytical processing. That means joining, selecting and aggregating through comprehensive tables. It is not important to keep data always up to date in this type of database. On Figure 2.1, you can see levels of hierarchy in companies.

2.2.1 On-line Transaction Processing (OLTP)

OLTP is optimized for large number of short on-line transactions. Data is operational and stored in original source. The main emphasis put on these systems is maintaining data integrity in multi-access environments and very fast query processing. Tuples are arranged in rows which are stored in blocks. Typical usage is concurrent inserts and updates, full row operations and simple queries with small result sets, used to create sales orders, invoice, accounting documents or to display customer master data. OLTP is represented by bottom layer of pyramide (Transaction Processing Systems) on Figure 2.1. [7]

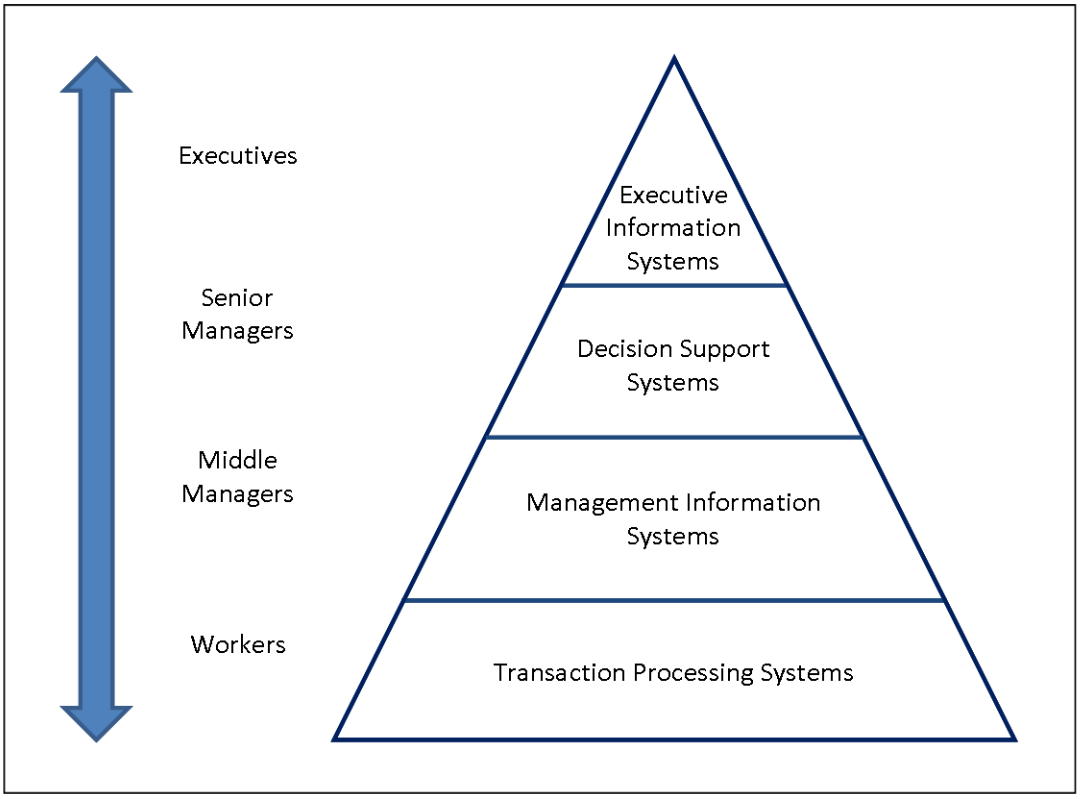

Executives
Senior Managers
Middle Managers
Workers
Executive Information Systems
Decision Support Systems
Management Information Systems
Transaction Processing Systems

Figure 2.1: A four level pyramid model of different types of Information Systems based on the different levels of hierarchy in organization (source: www.en.wikipedia.org)

2.2.2 On-line Analytical Processing (OLAP)

OLAP system is adapted for analytical processing of large amounts of data. It comes into system as predefined subset of the transactional data. Data is reproduced at regular intervals which causes the occasional outdated information for very complex queries, involving aggregations. The downside of OLAP is also very frequent data redundancy. The most common use is planning, forecasting and reporting, practically for operational reporting (list open sales orders, accounts receivables) or trial balance. OLAP is represented by top three layers (Management Information Systems, Decision Support Systems, Executive Information Systems) on Figure 2.1. We can also think of the data in a fact table as being arranged in a multidimensional cube. Very common in OLAP applications is storing data in a star schema with a fast table in the center and dimension tables radiating from it. In OLAP systems, new operations like drill down or roll up can be performed. [7]

2.3 Examples of new requirements

Our systems can store a lot of data, but their processing is really time consuming. Our applications collect structured data from sensors, web pages or information systems and unstructured data from end users. People feed our datacenter with their photos, feelings or just when browsing the web, our web browser stores information for next analysis.

2.3.1 Sensors

For sensors data catching, I found very nice example in a book from prof. Plattner [8]. This small example is about monitoring Formula 1 car performance. Mechanics need to know a lot of information about technical condition of tires, breaks, shock absorbers or engine. Each car has about 600 sensors and each sensor records multiple events per second. Each event means one insert to database. You are the person, who is responsible for car during the race. Your task is to command the driver to go to pit stop to fill fuel or change tires. What you really need is real time analytics of the data provided by car. If the response time of system take more than few seconds, you can use it only for analyses after the race.

2.3.2 Combination of Structured and Unstructured Data

Data is structured, if we can store them in a format, which is automatically processed by computers. Structured data is represented by tables, arrays or tree structures. Secondly, unstructured data cannot be easily analyzed automatically. Music, videos, photos or documents can be stored in a text format in databases, with meta data for searching purposes. With in-memory database technology, we will be able to process unstructured data without meta data and to search entire document. [8]

2.3.3 Social Networks and Search Terms

Very popular web pages are social networks. They started with sharing activities and photos between friends. Then the word „friend“ has changed its meaning and nowadays they are used for branding, recruiting and marketing. Another example for extracting business relevant information from the Web is monitoring search terms. The search engine Google analyzes regional and global search trends and according to the results, it can recognize for example symptoms of epidemic.

2.4 Combining OLTP and OLAP

Today's data management systems are optimized either for transactional or analytical workloads. Professor Hasso Plattner with his team checked several existing systems, customer systems with data, focused primarily on workload of write and read operations [8]. In Figure 2.2 you can find the result of their research.

When you go into the details, you will find out the systems usage is very similar, so there is no justification for different databases. The main benefit of the combination is that transactional and analytical queries can be executed on the same machine using the same set of data. The greatest benefit is undoubtedly the time spent for copying data from one database type (OLTP) to another (OLAP). If data is stored in one place, there is no need to handle how to replicate data between two system types. This combination will provide the management with current data and allow us to instantly analyze the data in the database and send a proper set of information to end users. The redundancy of data is very low, because we have only one database. We do not need complicated ETL process, used for copying data from OLTP to OLAP.

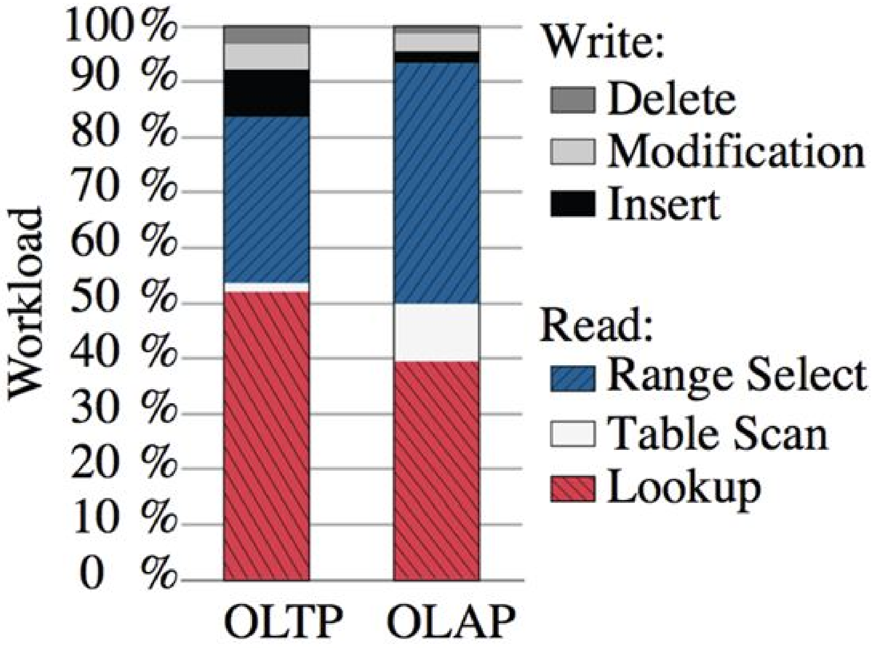


Figure 2.2: OLTP and OLAP accessing [8]

2.5 Hardware Changes

For this thesis is very important to have brief overview of the changes made in hardware. Our modern CPU can digest 2 MB of data per millisecond [8]. With ten cores, we can scan 20 MB per millisecond and if there are ten nodes per core, then we get already 200 GB per second scanning speed. For this hardware configuration, it is very difficult to write an algorithm which needs more than one second for execution. In Figure 2.3 the basic knowledge about memory hierarchy is described. The slowest part of our memory is hard disk. It is very cheap and offers a lot of space for data. Flash memory is faster than hard disk seek operations, but it works on same software principles as hard disk. That means the same block oriented methods which were developed more than 20 years ago for disks can be still used in a flash disk. Next part of pyramid is the main memory, which is directly accessible. This memory has reached rapid development in last years and now it can provide more storage space. The penultimate layer is reserved for the CPU caches L3, L2, L1 with different characteristics. From L1 cache, information goes directly into CPU register. All operations, like calculating are carried out here.

The bottleneck of the current approach are all hard disk operations. In this context, bottleneck means the layer of the pyramid on Figure 2.3, where transaction data is delayed most. Let me mention latency numbers of each layer [8]:

- **L1 cache reference:** 0.5 nanoseconds
- **Main memory reference:** 100 nanoseconds
- **Disk seek:** 10,000,000 nanoseconds

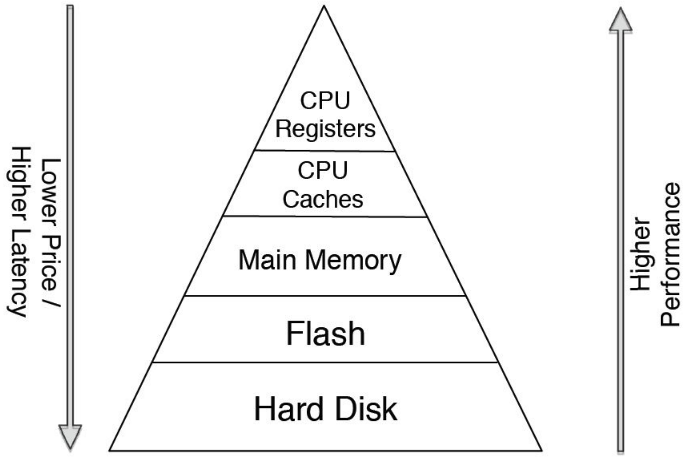


Figure 2.3: Memory hierarchy [8]

Those three numbers are very important. Using simple calculation we find out, that if we store data in main memory only and main memory will be our new bottleneck, we can speed up our data processing with factor 100,000. With this innovation, we have to cope with fact, that main memory is volatile. If we unplug the power, we lost all data. This is solved with non-volatile memory, most often with SSD disk, where we can store passive data (history) and snapshots for logging and recovering. Those concepts will be described in the subsequent sections.

The speed of networks for connecting nodes increases. Current nodes are connected in general with 10 Gb Ethernet network, but on the market we can find 40 Gb connections and switch manufacturers are talking about 100 Gb Infiniband. [8] They develop also switching logic allowing smart computations, improving joins of tables, where calculations often go across multiple nodes.

Chapter 3

Architecture

In this chapter, the SanssouciDB is described. It is a prototypical database system for unified transactional and analytical processing and SAP HANA database is based on knowledge of this prototype. [8] This concept was developed at the Hasso Plattner Institute in Potsdam as an SQL database and it contains similar components as other databases. Query builder, meta data, plan executer, transaction manager, etc. Understanding this prototype is important to explore how databases can become more complex and faster. Changes are mainly in storage technique, data layout in memory and distribute data to active and passive section. The end of this chapter contains brief overview, how logging and recovery works and what means merge operation.

3.1 Data Storage in Main Memory

Todays databases mostly use disks to store data and a main memory to cache data before processing them. In SanssouciDB, data is kept permanently in a main memory, but we still need non-volatile data storage for logging and recovery which is discussed at the end of this chapter. Using main memory as the primary storage leads to a different organization of data that only works if the data is always available in memory. For this case, all what is necessary is pointer arithmetic and following pointer sequences to retrieve data.

3.2 Data layout in main memory

Data stored in tables, we can simply imagine as two-dimensional, but memory in todays computers has still linear address layout. In this section, I will consider three ways, how to represent table in memory using row layout, columnar layout and a combination of these, a hybrid layout.

3.2.1 Row Data Layout

In a row data layout, data is stored tuple-wise. This brings low cost for reconstruction, but higher cost for sequential scan of a single attribute. In Figure 3.1, you can see how data is accessed for row and column based operations. This approach is always faster when we need all tuples, but - as mentioned below - in enterprise scenario we usually use only few attributes of tables.

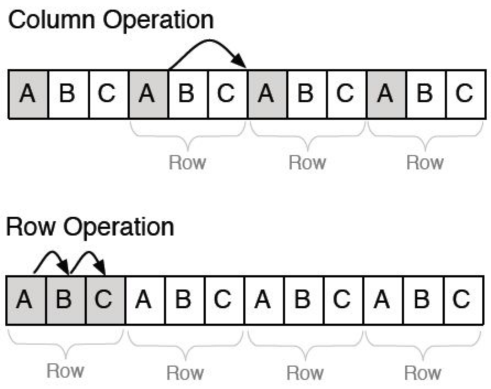


Figure 3.1: Memory accesses for row data layout [8]

3.2.2 Columnar Data Layout

When we use columnar data layout, data is stored attribute-wise, as you see in Figure 3.2. For this approach is typical acceleration of sequential scan-speed in main memory, but when we reconstruct tuples, it becomes expensive. There are many advantages speaking in favor of columnar layout usage in an enterprise scenario.[8] For example, using the columnar layout allows us to use dictionary encoding in combination with other techniques. It also enables very fast column scans allowing on the fly calculations of aggregates, so we do not need pre-calculated aggregates in the database anymore. This will minimize redundancy in database.

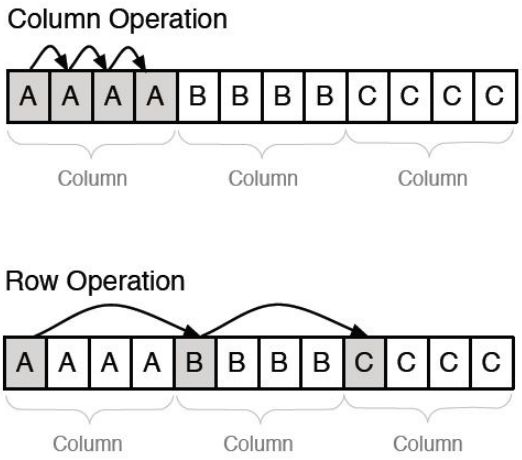


Figure 3.2: Memory accesses for column data layout [8]

3.2.3 Hybrid Data Layout

Hybrid data layout is a combination of previous layouts. This layout depends mostly on specific table requirements. The main idea of having columnar and row layout mixed is that if the set of attributes are processed together, it makes sense to store them physically together. Advantage of using this layout depends greatly on the programmer. For better understanding see Figure 3.3.

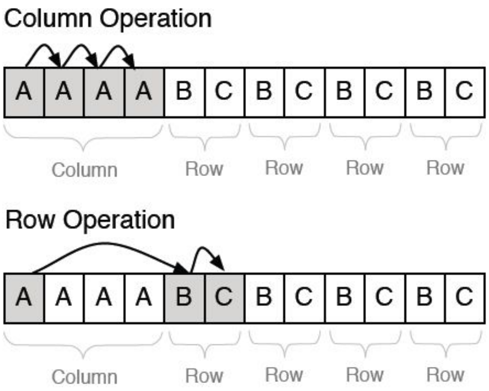


Figure 3.3: Memory accesses for hybrid data layout [8]

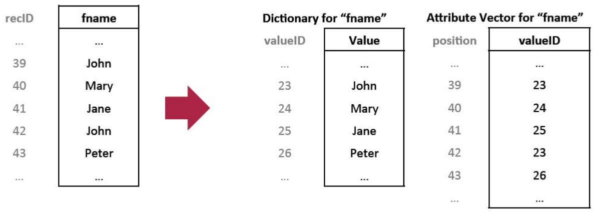
3.3 Consequences of Column Orientation

Column oriented storage has become used in specific OLAP database systems. An advantage of this data layout is clear in case of sequential scanning of single attributes and set processing. Enterprise application analysis showed that there is actually no application using all fields of a tuple. For example, in dunning only 17 needed attributes are queried instead of the full tuple represented by 300 attributes. For programmers, it is very important to avoid „SELECT *“ statement and focus on selecting only minimal set of data needed for application. The difference in runtime between selecting specific fields or all tuples in row oriented storage is insignificant, but in case of column orientation storage, the penalty for „SELECT *“ statement is very expressive.

When a database stores data in columns instead of rows, it is more complicated for write access workloads. Due to this issue, the differential buffer was introduced. This differential storage is optimized for inserts and data is merged into main store in regular intervals. More details about the differential buffer and merge process is provided in section 3.6.

With the column store, we can reduce the number of database indices, because every attribute can be used as an index (integer value) and the scanning speed is still high enough. For further speedup, dedicated indices can still be used.

3.4 Compression of data in memory



recID	fname
...	...
39	John
40	Mary
41	Jane
42	John
43	Peter
...	...

Dictionary for “fname”

valueID	Value
...	...
23	John
24	Mary
25	Jane
26	Peter
...	...

Attribute Vector for “fname”

position	valueID
...	...
39	23
40	24
41	25
42	23
43	26
...	...

Figure 3.4: Example of Dictionary Encoding [8]

If we want to store data in main memory, we have to consider the size and how we need to process them. By analyzing enterprise data, special data characteristics were identified. Most interestingly, many attributes of a table are not used and table can be very wide.

55% of columns are unused on average per company and tables with up to hundreds of columns exist [8]. Many columns that are used have a low cardinality of values. Further, in many columns NULL or default values are dominant, so the entropy (information containment) of these column is very low. To reduce size of this type of data we can use Dictionary Encoding. [8] Dictionary Encoding is based on a very simple principle. Each column is split into a dictionary and attribute vector. In dictionary, we store all distinct values with their valueID. Attribute vector contains only valueIDs, which correspond to the valueIDs in the dictionary. This brings possibility to store all information as integers instead of other, larger data types, and to perform operations directly on compressed data. Example of Dictionary Encoding can be seen in Figure 3.4. To achieve better performance, it is necessary to sort data in dictionary. Then a binary search is possible to find values, but we have to change principle of inserting data as discussed in the next chapter.

For better understanding of this very important basics of reducing data size, I choose an example of world population with 8 billion tuples from [8]. Lets look in detail in Figure 3.5.

For column Last names, we need to represent 8 million distinct values.

$$\log_2(8milion) = 23$$

Size of attribute vector itself can be reduced to

$$8billion \cdot 23bit = 21.42GB.$$

Dictionary needs to reserve

$$50Byte \cdot 8million = 0.38GB.$$

Plain size can be calculated like

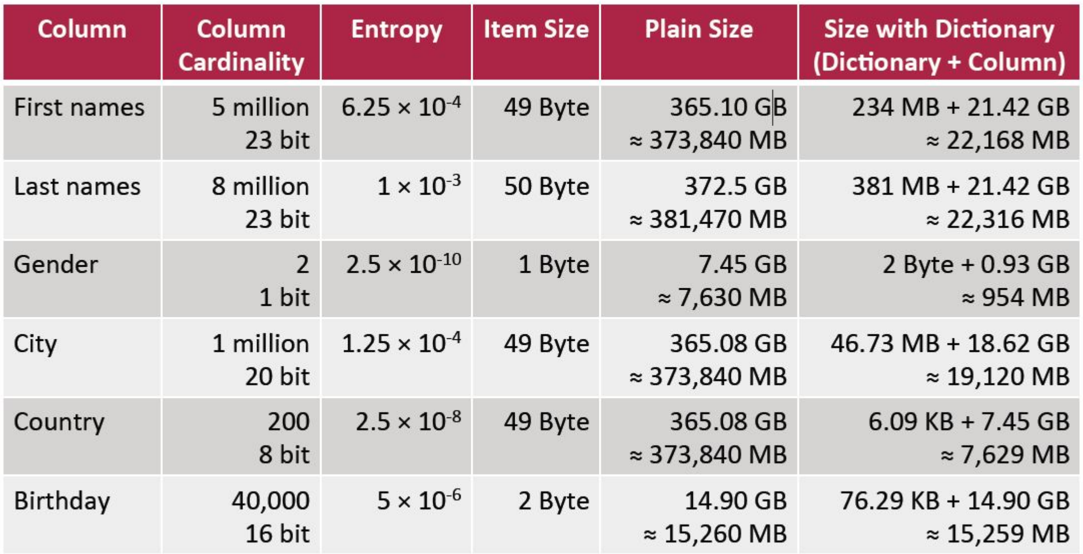
$$8billion \cdot 50Byte = 372.5GB.$$

Now, we can calculate compression factor

$$\frac{uncompressed\ size}{compressed\ size} = \frac{372.5\,GB}{21.42\,GB + 0.38\,GB} \doteq 17$$

With Dictionary Encoding, we need only 6% of the initial amount of main memory. Regarding Figure 3.5, we can calculate next attributes in our table in the same way. Total plain size is 1,525,880 MB. Size of Dictionary encoded data is 87,446 MB. Dividing these numbers, we get factor of compression 17 for entire table. This helps us to use main memory as a storage for our data. Compression rate depends on the size of the initial data type and on column's entropy - a measure which shows how much information is contained in column. Column entropy can be calculated as ratio of column cardinality and table cardinality.

- **Column cardinality** is number of distinct values in one column.
- **Table cardinality** is number of tuples in a relation.



Column	Column Cardinality	Entropy	Item Size	Plain Size	Size with Dictionary (Dictionary + Column)
First names	5 million 23 bit	6.25 × 10^{-4}	49 Byte	365.10 GB ≈ 373,840 MB	234 MB + 21.42 GB ≈ 22,168 MB
Last names	8 million 23 bit	1 × 10^{-3}	50 Byte	372.5 GB ≈ 381,470 MB	381 MB + 21.42 GB ≈ 22,316 MB
Gender	2 1 bit	2.5 × 10^{-10}	1 Byte	7.45 GB ≈ 7,630 MB	2 Byte + 0.93 GB ≈ 954 MB
City	1 million 20 bit	1.25 × 10^{-4}	49 Byte	365.08 GB ≈ 373,840 MB	46.73 MB + 18.62 GB ≈ 19,120 MB
Country	200 8 bit	2.5 × 10^{-8}	49 Byte	365.08 GB ≈ 373,840 MB	6.09 KB + 7.45 GB ≈ 7,629 MB
Birthday	40,000 16 bit	5 × 10^{-6}	2 Byte	14.90 GB ≈ 15,260 MB	76.29 KB + 14.90 GB ≈ 15,259 MB

Figure 3.5: Definition of example table of the world population [8]

3.5 Active and Passive Data

The data in SanssouciDB is separated into active and passive data. Passive data is stored in a slower storage and queried less frequently. Active data is stored in main memory and this distribution of data reduces the amount of main memory needed to store the entire data set.

Storing data in main memory has one very nice feature. Main memory accesses depend on time deterministic processes in contrast to seek time of disk, that depends on mechanical parts. This mean, that response time of in-memory database is smooth, always the same, and response time of disk is variable because of disk seeks.

3.6 Architecture Overview

The architecture shown in Figure 3.6 provides an overview of the components of SanssouciDB. This database is split into three logical layers. The Distribution Layer provides the communication with applications, Metadata, and creates Query Execution plans. The Main Store, Differential Store and Indexes are located in a Main Memory layer at Server. Content of the third layer: Logs, Snapshots and Passive Data are stored in a Non-Volatile memory. All these concepts will be described in subsequent sections.

3.6.1 Differential buffer and Merge

When a new tuple comes in, it does not go into the main storage, but into a differential buffer. In this buffer we also keep data in a column store format, but its dictionary is much smaller then main storage dictionary and it does not need to be resorted, so inserting into differential buffer is always faster. All insert, update and delete operations are performed on the differential buffer. During query execution, a query is logically split into a process in the compressed main partition and in the differential buffer. After the results of both

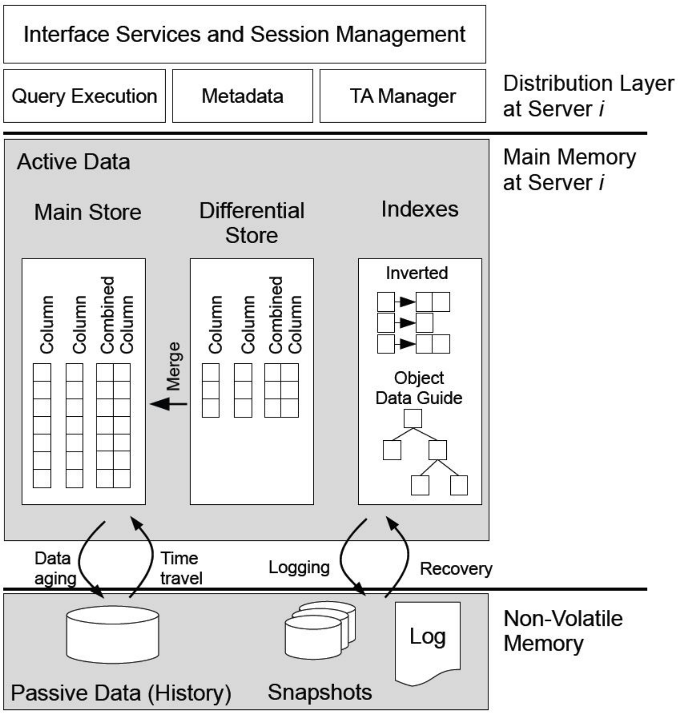


Figure 3.6: Schematic architecture of SanssouciDB [8]

subsets are retrieved, they must be combined to build a full valid result representing the current state of the database.

To guarantee optimal performance, the differential buffer needs to be merged periodically with the main store. This process has three phases. Preparation, attribute merge and commit. For better understanding of merge operations, you can see the details in Figure 3.7. Before starting the merge operation, we have main store and differential buffer for data modifying operations. When we start the merge process, we have to create new main store and new differential buffer, because the system is still prepared for another read or insert operation and we achieve asynchronous merge. In the last step, the commit phase, our main memory contains new main store and new differential buffer with operations, that come during second step of merge.

3.6.2 Logging and Recovery

Databases need to provide durability guarantees (as part of ACID[1]) to be used in productive enterprise applications. To provide these guarantees, fault-tolerance and high availability have to be ensured. The standard procedure to enable recovery is logging. Data is written into log files, which are stored in a persistent memory. Infrastructure of SanssouciDB logging has three parts: snapshot of main store, value logs and dictionary logs. Snapshot is ideally written after each merge process and changes in the differential buffer are logged in log files. Values and database dictionary are stored in different files because we can replay it in parallel and not in one sequence. In addition to logging, SanssouciDB logs meta data to speed up the recovery process.

[1] ACID is Atomicity, Consistency, Isolation, Durability. These properties guarantee reliability for database transactions and are considered as the foundation for reliable enterprise computing.

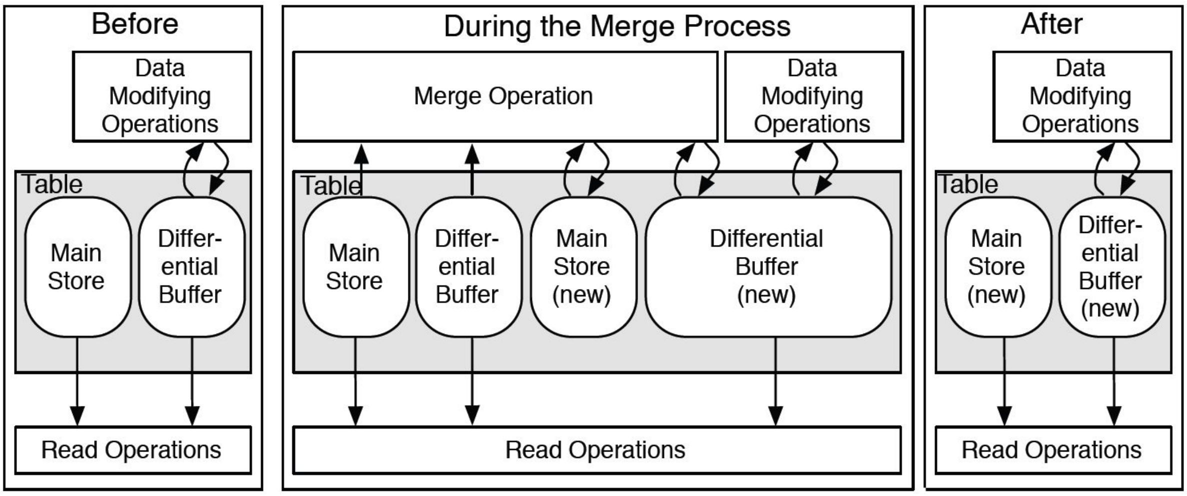


Figure 3.7: The concept of merge operation [8]

When a server fails, it may have to be rebooted and restored. It can be caused by blackout or internal issue on server. When the recovery of our system starts, we need first to read meta data for table sizes, latest snapshots etc. After we gather all needed information, we can perform next steps in parallel processing:

- Recover main store from snapshot
- Recover dictionary of delta store by replaying dictionary logs
- Recover valueID vectors of delta store

After the import of log files, the second run over the imported tuples is performed. This is caused by the dictionary encoded logging, which only logs changed columns. At the end, the imported tuples have to be checked for empty attributes and they have to be completed. This is done by iterating over all versions of the tuple, as recorded in a validation flag.

3.7 SAP HANA Platform

SAP HANA platform is not only in-memory database based on SanssouciDB prototype, called HDB. This platform also contains an Eclipse based tool developed especially for SAP HANA development, called SAP HANA Studio. This tool is very useful for creating applications on SAP HANA, modeling and visualizing the data. I did not use this tool to create my application, because I need to get access into MaxDB.

Chapter 4

Basic operations

In this chapter, basic database operations in SAP HANA database are described in detail. Storing the data in columns is also related to several changes in the implementation of database management. Process used to provide the same data interfaces as known from row stores in column store is called materialization and it is described at the end of chapter. For introduction to the basic principles of SAP HANA I also recommend the book [4]. Basic characteristics of SAP HANA is also well described in article [6].

4.1 Database Operators

Due to the use of dictionary encoding described in 3.4, the implementation of individual operations must be changed. See description below, how each operator works.

4.1.1 `DELETE`

`DELETE` operation basically cancel the validity of given tuple. I.e. it allows us to say that certain item in the database is no longer valid. This operation can either be the physical or logical operation. The physical deletion removes a tuple from the database and it cannot be retrieved anymore. The logical deletion only terminates the validity of this item in the dataset, but the tuple is still available and could be used for example for historic queries. Example of `DELETE` operation can be explained with Figure 4.1.

In this case, we want to delete Jane Doe from database. If we look at our dictionary encoded table, we can see that we have dictionary for first name, for last name and corresponding attribute vectors. First we have to identify valueID for Jane (23) and Doe (18). Now we have to look at the attribute vector with position of these values. For this example it is recID 41, which id deleted and all subsequent tuples are adjusted to maintain a sequence without gaps. This makes the implementation of deletion very expensive. The value remains in the dictionary for furher use.

4.1.2 `INSERT`

Compared to row oriented database, the insert in a column store is a bit more complicated. In row oriented database, new tuple is simply appended to the end of the table. Adding a new tuple to a column means to check the dictionary and add a new value if necessary. Afterwards, the respective value of the dictionary entry is added to the attribute vector. Considering that the dictionary is sorted, adding a new tuple has three different scenarios.

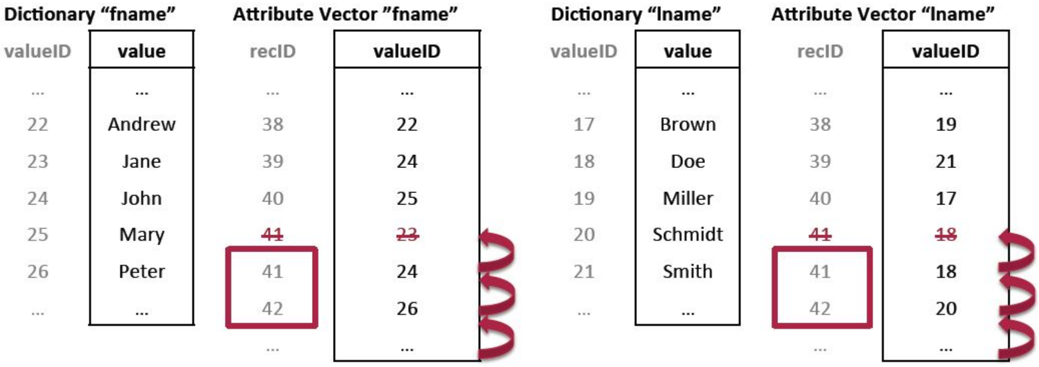


Figure 4.1: Example of DELETE operation. [8]

- INSERT **without New Dictionary Entry** The best situation occurs, when our dictionary already contains inserted value. In this case, we simply append its valueID at the end of attribute vector with maintaining the sequence.
- INSERT **with New Dictionary Entry without resorting the dictionary** In this case, value is appended into dictionary and rest of the operation is similar to the previous case.
- INSERT **with New Dictionary Entry with resorting the dictionary** Regading Figure 4.2, we want to add value Karen into column. For this example, we have to perform following steps:
  1. Append value Karen at the end of our dictionary.
  2. Re-sort the dictionary.
  3. Change all valueIDs in attribute vector.
  4. Append new valueID to new attribute vector.

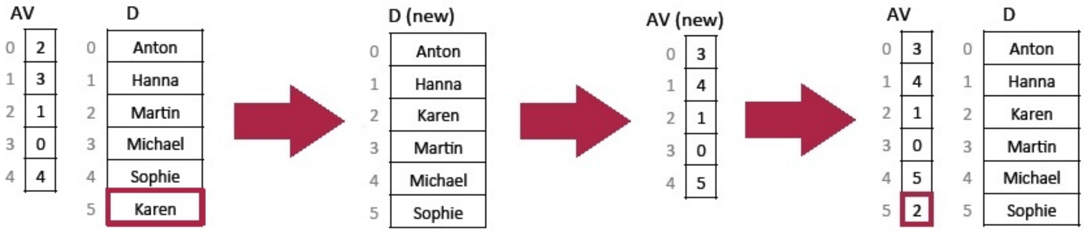


Figure 4.2: Example of INSERT operation. [8]

4.1.3 SELECT

In most applications, SELECT is a commonly used command. It presents a declarative description of the result requested from database. To extract the data from the database, an ordered set of execution steps is required. It is called query execution plan. For each query, multiple execution plans can exist that deliver the same results, but with different performance. To calculate cost of different query execution plans, query optimizers are used. Task of the query optimizer is to choose the most efficient plan. The main goal is to reduce the size of the result set as early as possible. This goal can be achieved by applying selections as early as possible, with ordered sequential selections (most restrictive selections are executed first) or with ordered joins corresponding to their tables (smallest tables are used first).

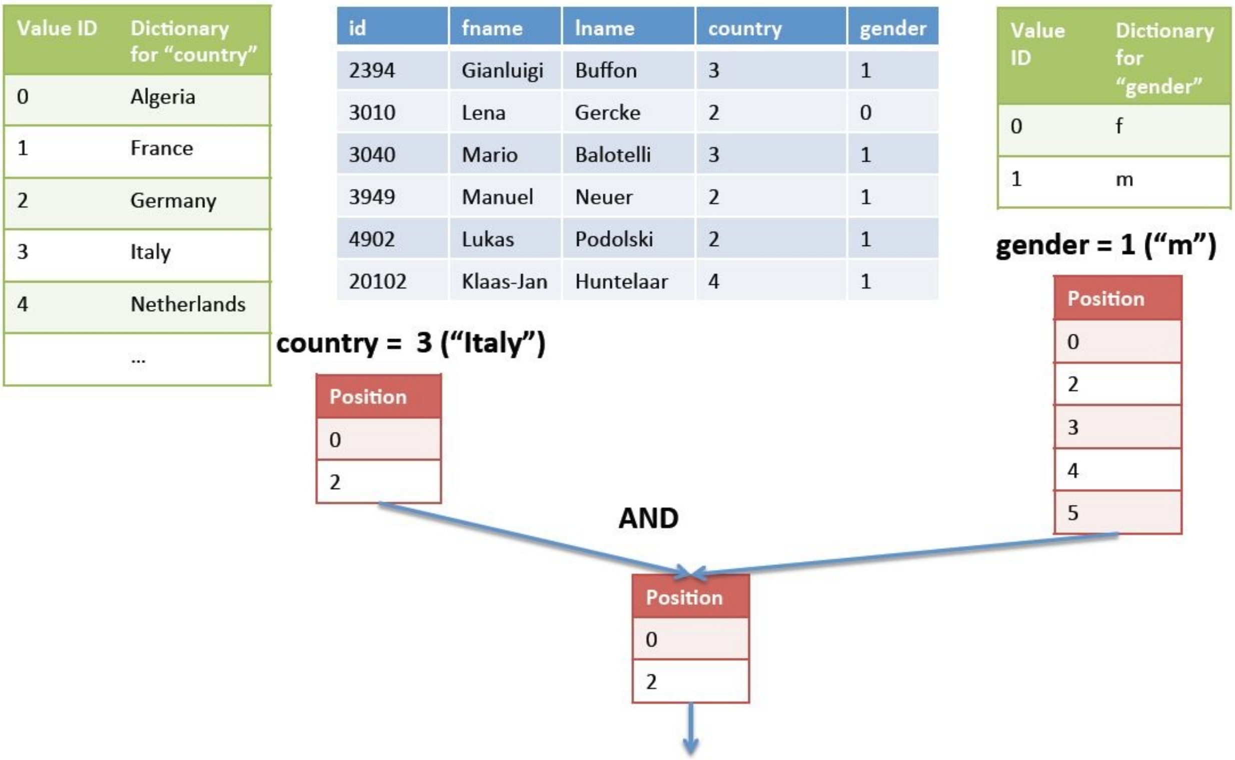
Figure 4.3 shows execution of statement `SELECT fname, lname FROM population WHERE country = 'Italy' AND gender = 'm'`. This operation is executed in the following steps:

1. Parallel scan of country and gender dictionaries for ValueIDs.

2. Parallel scan of attribute vector for positions.

3. Logical AND operation on positions of both country and gender attributes.

4. Final list of positions that we want to select.



Value ID	Dictionary for "country"
0	Algeria
1	France
2	Germany
3	Italy
4	Netherlands
	...

id	fname	lname	country	gender
2394	Gianluigi	Buffon	3	1
3010	Lena	Gercke	2	0
3040	Mario	Balotelli	3	1
3949	Manuel	Neuer	2	1
4902	Lukas	Podolski	2	1
20102	Klaas-Jan	Huntelaar	4	1

Value ID	Dictionary for "gender"
0	f
1	m

Position
0
2

Position
0
2
3
4
5

Position
0
2

Figure 4.3: Example of `SELECT` operation. [8]

This example can be executed also sequentially. In this case it is always better to get subset of values by scanning column country and then scan this subset for column gender, because gender has only two distinct values.

4.2 Materialization strategy

Part of SELECT operation is materialization in order to provide the same data interfaces as known from row stores. The returned results have to be transformed into tuples in a row format. For this operation we know two different materialization strategies. Early and late materialization. Both of them can be superior depending on the storage techniques.

In a nutshell, early materialization decompresses and decodes data early and then operates with strings (in this case), while late materialization operates on integers and columns as long as possible. Very illustrating comparison of both strategies can be shown in Figure 4.4 for statement `SELECT city, COUNT(*) FROM population WHERE gender = 'm' AND country = 'GER' GROUP BY city`. For dictionary encoded, columnar data structures, late materialization strategy is mostly used. In **early materialization** strategy, all required columns are materialized first. For our example, in the left side of Figure 4.4 we have to perform these steps:

1. Scan for constraint on gender.
2. Scan for constraint on country and add it to the result.
3. Look up city values in dictionary.
4. Add the city attribute vector to the result.
5. Group and count.

In **late materialization** strategy, the lookup into the dictionary for materialization is performed in the very last step before returning the result. In the right side of Figure 4.4, you can follow the steps of late materialization:

1. Look up constraint values in dictionary.
2. Scan for constraint in attribute vector.
3. Logical AND.
4. Filter attribute vector by GROUP BY.
5. Look up values from the dictionary.

4.3 Hash Join

The hash-join is typical join known from relational databases, based on a hash function, which allows access in a constant time. This function allows to map variable length values to fixed length keys. The hash join algorithm consists of two phases: hash phase and probe phase. During the hash phase, the join attribute of the first join table is scanned and a hash map is produced where each value maps to the position of this value in the first table. In the second phase, the second join table is scanned on the other join attribute and for each tuple, the hash map is probed to determine if the value is contained or not. If the value is contained, the result is written.

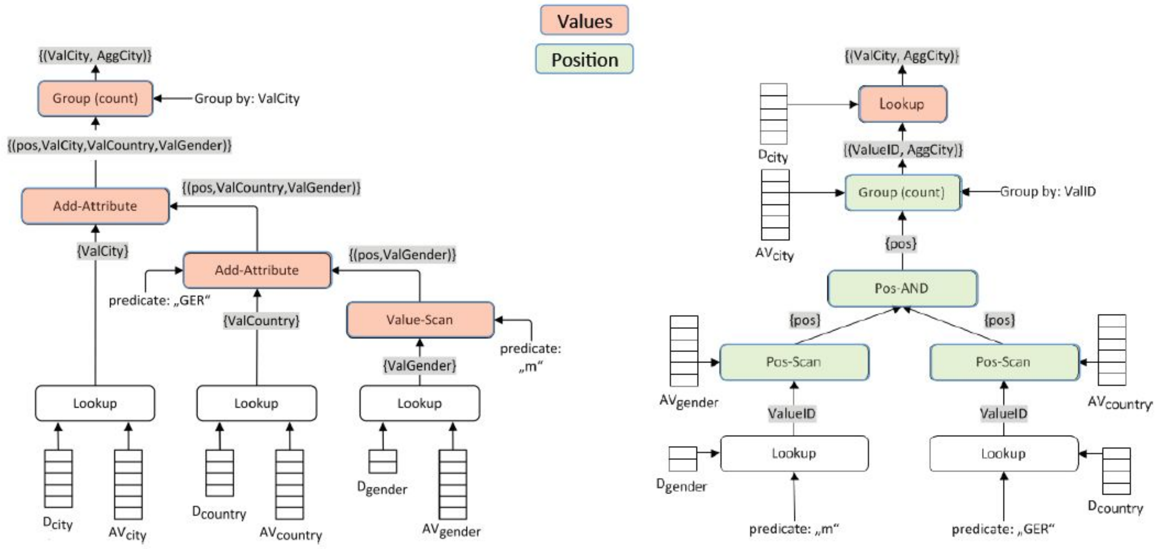


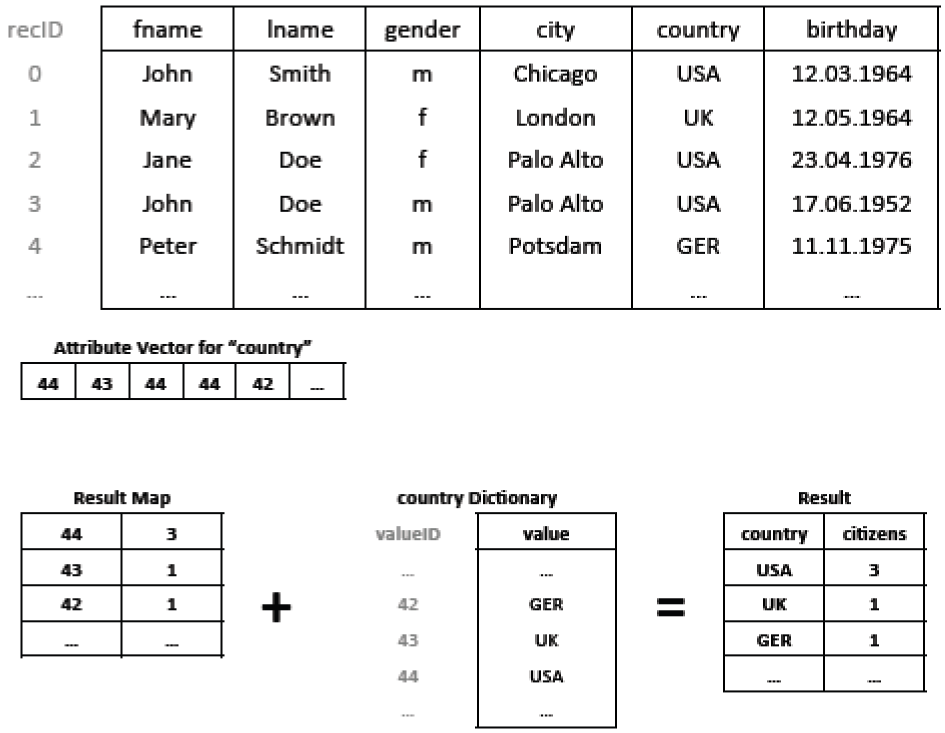
Figure 4.4: Comparison between early and late materialization. [8]

4.4 Aggregation Example

Let us consider an example for the use of the `COUNT` function. Input table and steps of execution are included in Figure 4.5. The goal is to count all citizens per country using this `SELECT` statement:

```
SELECT country, COUNT( * ) AS citizens
   FROM world_population
   GROUP BY country
```

First, the system runs through the attribute vector for the country column. For each new encountered valueID, an entry with initial value „1“ is added to the result map. If the encountered valueID already exists, the entry is incremented by one in the result map. After this process, the result map contains the valueIDs of each country and its number of occurrence. Finally, the result is materialized. The countries are fetched from the dictionary using the valueIDs and the final result of the `COUNT` function is created. Final result of this operation is table which contains a pair of country names and the number of occurrences in the source table, which corresponds to the number of citizens. This pattern is similar for other aggregate functions (`SUM, AVG, MAX, MIN`) [8].



recID	fname	lname	gender	city	country	birthday
0	John	Smith	m	Chicago	USA	12.03.1964
1	Mary	Brown	f	London	UK	12.05.1964
2	Jane	Doe	f	Palo Alto	USA	23.04.1976
3	John	Doe	m	Palo Alto	USA	17.06.1952
4	Peter	Schmidt	m	Potsdam	GER	11.11.1975
...	...	...	...		...	...

Attribute Vector for "country"

44	43	44	44	42	...

Result Map

44	3
43	1
42	1
...	...

+

country Dictionary

valueID	value
...	...
42	GER
43	UK
44	USA
...	...

=

Result

country	citizens
USA	3
UK	1
GER	1
...	...

Figure 4.5: Aggregation example. [8]

Chapter 5

Test application

My task is to create simple test application to compare performance of standard SAP system running on usual database with HANA database. Thanks to cooperation with SAP, I have access to SAP performance systems as a developer. It allows me to create local database objects and develop my own executable program. My application is developed in ABAP programing language and the very inspiring literature to develop my application was [9] by James Wood. In this chapter I also describe implementation of the test application, source of data for testing and direction for use application. HDB is an acronym of SAP HANA database.

5.1 ABAP - Advanced Business Application Programming

ABAP is a programming language designed to develop applications for SAP system. From the last version, it is object-oriented programming language. ABAP has very similar programming constructions like COBOL or SQL, when working with database tables. To edit programs, ABAP Workbench is available where you can take care about your code. There are several types of programs:

- **Executable programs:**
  - **Reports:** Report is a single program, which takes some input parameters from user and shows results, mostly in a table view. In a report, you can also define screen with text fields, buttons or subscreens. I use this type of program for my application, because more complex programming is not required in my case.
  - **Module pools:** Module pools define more complex programming with screens and their flow logic.
- **Non-executable programs:**
  - **INCLUDE modules:** In INCLUDE, you can define part of code which is repeated; programmers use it to subdivide very large programs.
  - **Subroutine pools:** If you want to create a method, you can enclose your code in FORM/ENDFORM statements and if needed, invoke it by PERFORM.
  - **Function groups:** Function group is a kind of library of functions. Function code is enclosed in FUNCTION/ENDFUNCTION statements and executed with statement CALL FUNCTION. In a function group, you can define functions and

call them remotely from another system. This feature really helps me to have my application only on one system, with access to MaxDB system.

- **Object classes:** Object class is a set of methods and attributes. It is used for object oriented programming.
- **Interfaces:** Interface contains method definitions.
- **Type pools:** In a Type pool, you can define data types and constants.

5.2 SAPLINK and abap2xlsx

Saplink is an opensource project that provides transport of ABAP object between systems. With saplink you can choose which object you want to transport and program generates a text file in the xml format. You can upload this file to another system. It is very helpful, when you create local objects without using SAP transporting solutions. I use this tool to copy my objects between systems. Some programmers use this tool to share their ABAP developments. Saplink can be used for name refactoring of objects. [1]

Abap2xlsx is also an opensource project allowing you to create your own Excel spreadsheet. The source code can be installed via SAPLINK. I use this tool to export results of my test into Excel file for further processing.[2]

5.3 Used systems and RFC connection

To perform my tests, I got access to SAP internal development systems named ZKU and ZPU. ZKU is uses MaxDB, which is an ANSI SQL-92 compliant relational database management system from SAP AG. [5] ZPU is based on the HDB database, which is part of SAP Hana platform. On both systems, component version EHP7 is installed for SAP ERP 6.0. A test application can be run only from ZPU system.

RFC (Remote Function Call) is interface for communication between SAP Systems or with other external systems using TCP/IP or CPI-C connections. I use this interface to execute functions on ZKU system and to take over results in my application placed in system ZPU. On ZKU I created function group `ZMURFCS`, that contains required functions. On ZPU I defined RFC connection to ZKU and then I can call remote functions with parameter `DESTINATION`, which is my defined RFC connection. How RFC is used in test application can be seen in Figure 5.1.

5.4 Data

One of important things of my task is to find an appropriate source of data. To achieve relevant results, I need more than one table with millions of tuples. I tried to ask my colleagues from SAP responsible for performance, especially SAP Hana testing, but nobody could provide me this kind of testing data. So I had to find my own free source and create tables. Benefit for me is fact, that I can have exactly the same data on both testing systems.

As source of data for my application, I have chosen GeoNames geographical database. This database is available from www.geonames.org and you can download text file with basic information about cities around the world. This download was free of charge, because it is under a Creative Commons Attribution license. There is also possibility to use GeoNames web services to get relevant data. I use file `allCountries.zip` from download server. This

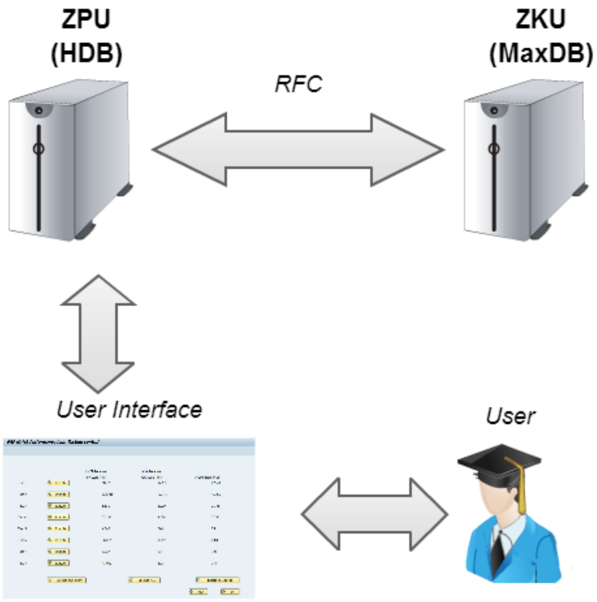


Figure 5.1: RFC connection in application.

file contains `allCountries.txt` with 8,567,146 tuples. Because I do not need so much tuples, I cut first 5,000,000 of tuples. I also use file timezones.txt with information about time zones.

Originally, the source database has 19 attributes. I decided to use only a few of them, because it contains lot of null values. The structure of the database that I use in experiments is shown in Figure 5.2. Field population contains originally lot of zero values so when my application is fetching data from file and population is null, it simply add random number. This allows me to test aggregate functions through all tuples. All attributes could be included in one table, but I need more tables to perform join operations ibnr my tests.

Table `ZMUCITY` has 7 attributes. For unique identification is attrbiute `ID`. `NAME` stands for name of the city, `COUNTRYCODE` identifies country in which the city is situated. Attribute `POPULATION` indicates the number of inhabitants, `ELEVATION` is distance above sea level, `MODIFDATE` is date of last modification and `TIMEZONEID` is refference value to table with informations about time zones called `ZMUTIMEZONES`. This table contains `COUNTRYCODE`, `GMT` (Greenwich Mean Time) and `DST` (Daylight Saving Time). In last table `ZMUPOSITION` the position informations `LATITUDE` and `LONGITUDE` are stored.

5.5 Implementation

In this section, I will describe implementation and how my application can be used. This application is written in ABAP language with standard ABAP Workbench with very useful debugging tool.

In a nutshell, this application contains four screens. First one, with controls of data in tables, second one with test execution controls - from this screen you can continue to execute own query or exporting data into Excel spreadsheet for further processing. Test application can be executed via transaction `SE38` (standard SAP transaction for ABAP Editor) with program name `ZMUTESTREPORT` from `ZPU`.

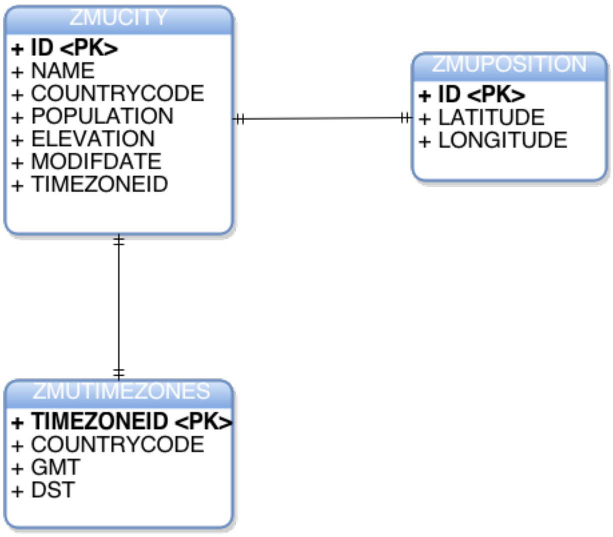


Figure 5.2: Structure of database

5.5.1 Getting data and insertion to database

On the first screen, captured in Figure 5.3, user can manage tables. Tables ZMUCITY and ZMUPOSITION are filled from one file and table ZMUTIMEZONE is filled from another file. That is the reason, why I create two rows. In the first row, there is an output field with number of rows of table ZMUCITY and two buttons. If you click on „Fill table“ button, you will be asked to find input file on your disk „Open“ button on this screen will start inserting the data into database. This screen contains also two display fields with duration of the last insertion. This time is also inserted into table ZMUINSERT after each fill table execution. Button „Empty table“ will start deletion of all tuples in table. In the second row, you can do the same operations for table ZMUTIMEZONES. There are also two navigation buttons „Next“, which will redirect you to next screen with test execution, and „Exit“, which you can use to finish your work with application.

To choose file from hard drive I use method file_open_dialog. This allows user to find the right destination of file to be uploaded. This destination is passed to method gui_upload. This method uploads file into the server for next processing. Both methods are part of class cl_gui_frontend_services.

5.5.2 Test execution

From the first screen, you can continue to the Test execution screen, which can be seen in Figure 5.4. For each test, there is a row with controls and text execution times. Only with „Execute ALL“ button, test results are saved in table ZMUSELECT. If you execute tests one by one, you cannot save it into Excel spreadsheet. For next navigation, four buttons are available. „Exit“ to exit application, „Back“ to show the first screen, „Execute own query“ can be used in case you want to type own query, and button „Export to Excel file“ switches to settings of the result export.

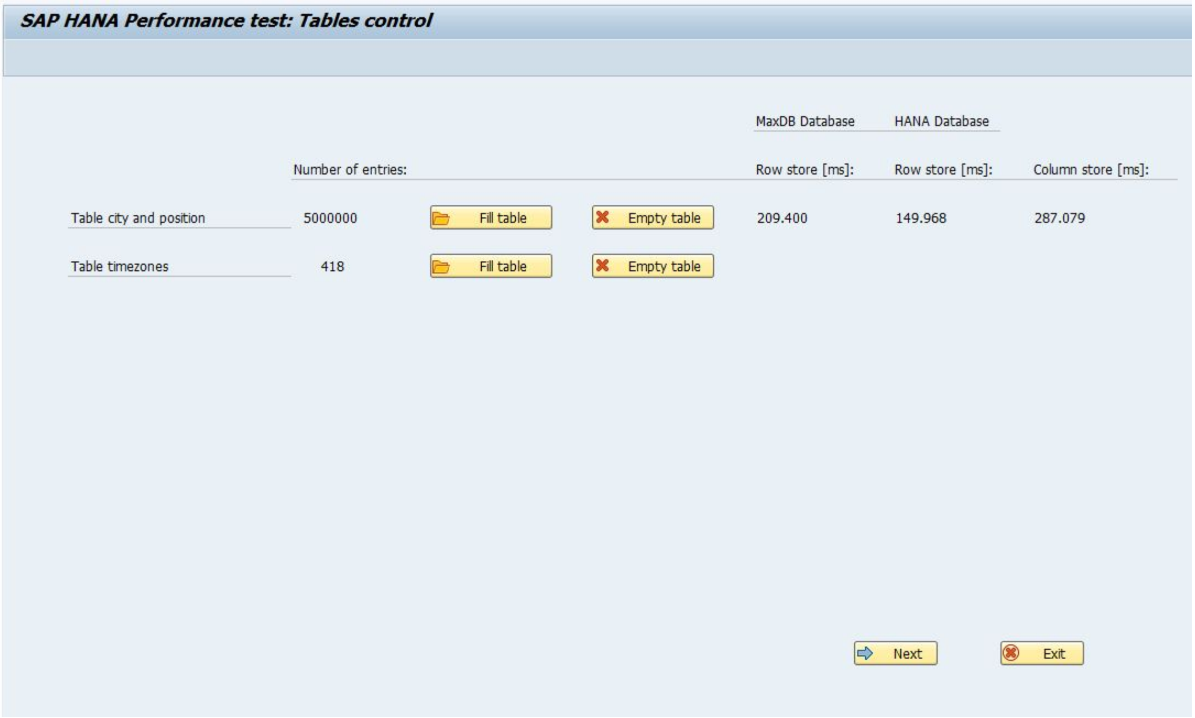
5.5.3 Exporting data

When users finish test execution, it is possible to export data into predefined Excel spreadsheet. Detail of this screen is in Figure 5.5. There are two ways to run this feature. Call transaction `SE38` and execute program `ZMUABAP2XLSX` or use navigation button from screen with test execution control. As input file in first input field, please choose file RESULTS_TEMPLATE.xlsx on your disk. In output options you can choose from actions below before you click on „Execute“ button.

- **Save to frontend** This option requires path on disk, where final document RESULTS.xlsx will be stored.
- **Save to backend** This option allows you to store document on application server. This file can be processed via standard SAP transaction `AL11`.
- **Direct display** If you want to check the file before saving, or sending via e-mail.
- **Send via email** Write down the e-mail of recipient into input field and file will be sent via e-mail.

5.5.4 Execute own query

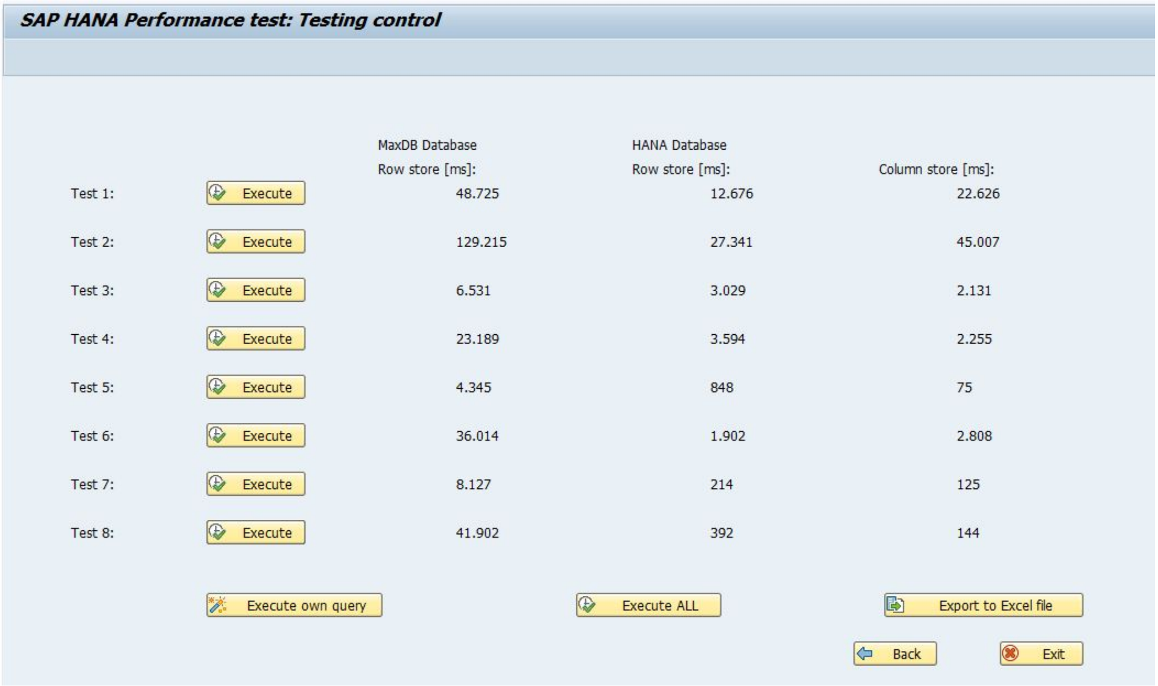
This screen is showed in Figure 5.6. The functionality can be accessed from test execution screen or via transaction `SE38`, program name `ZMUEXECQUERY`. Into input field, you can type your own dynamic SQL query, for example `SELECT * from ZMUCITY` and after execution, you will see execution times. In this report, object oriented programming is used, because I need to use existing ABAP methods to work with own query. Here is also handled exception of query execution, so if user makes a mistake in his/her query, error message is raised.



SAP HANA Performance test: Tables control

		MaxDB Database	HANA Database	
	Number of entries:	Row store [ms]:	Row store [ms]:	Column store [ms]:
Table city and position	5000000	209.400	149.968	287.079
Table timezones	418			

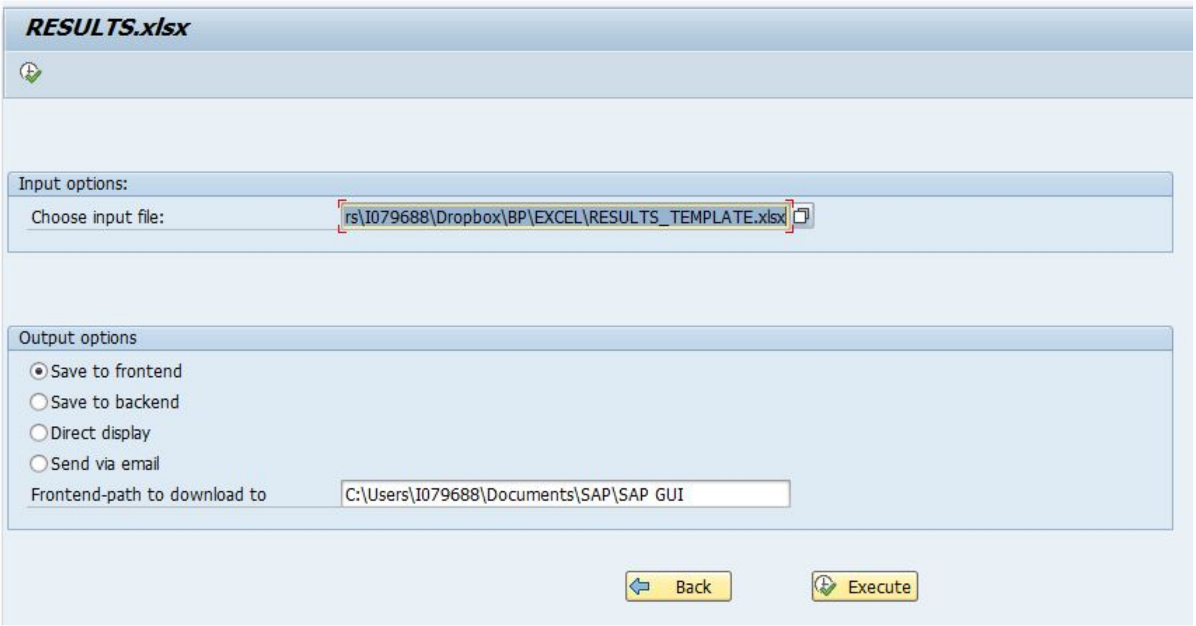
Figure 5.3: First screen of application.



SAP HANA Performance test: Testing control

	MaxDB Database	HANA Database	
	Row store [ms]:	Row store [ms]:	Column store [ms]:
Test 1:	48.725	12.676	22.626
Test 2:	129.215	27.341	45.007
Test 3:	6.531	3.029	2.131
Test 4:	23.189	3.594	2.255
Test 5:	4.345	848	75
Test 6:	36.014	1.902	2.808
Test 7:	8.127	214	125
Test 8:	41.902	392	144

Figure 5.4: Test execution control screen.



RESULTS.xlsx

Input options:

Choose input file: rs\I079688\Dropbox\BP\EXCEL\RESULTS_TEMPLATE.xlsx

Output options

● Save to frontend

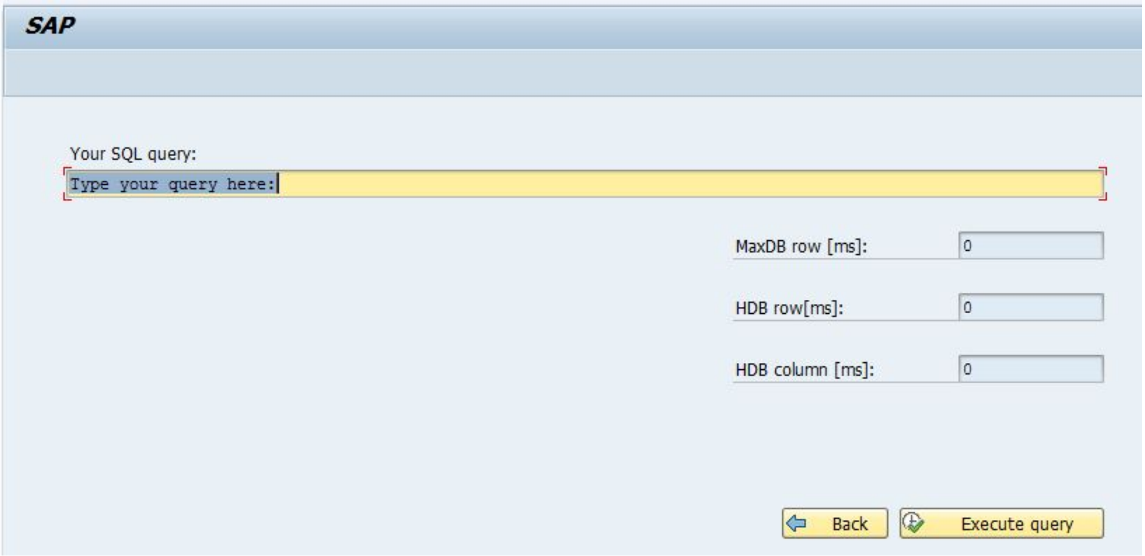
○ Save to backend

○ Direct display

○ Send via email

Frontend-path to download to C:\Users\I079688\Documents\SAP\SAP GUI

Figure 5.5: Data export screen.



SAP

Your SQL query:

Type your query here:

MaxDB row [ms]: 0

HDB row[ms]: 0

HDB column [ms]: 0

Figure 5.6: Execute own query screen.

Chapter 6

Test results

In this chapter, I describe my experiments. At first I focused to insertion time into different types of database. Main part of my testing was `SELECT` operation. First four tests are focused to row operations and next four tests are focused to column operations with aggregate functions. How much space consumes each table is also presented in this chapter.

To upload 5,000,000 tuples into tables, I split source file into ten parts with 500,000 tuples each. These files were uploaded one by one, so I got ten execution times.

Then I ran all tests (Test 1 - Test 8) also ten times, because in this chapter I discuss stability differences betweeb MaxDB and HDB.

At the end of testing, I exported data into predefined Excel spreadsheet. For comparing values, I used the average value of each measurement. For each test I also captured execution plan, because it helps me to explain differences between type of databases or storage types. Execution plans are captured from HDB system, first one for row layout and second one for column layout. Each test is executed on different systems or with different storage type:

- MaxDB database with row store layout.
- HDB database with row store layout.
- HDB database with column store layout.

In system with MaxDB database, it is also possible to select a checkbox to create table with column store, but this checkbox has no influence to layout type changes. I think, this could be confusing for programmers.

6.1 Size of tables

Chart in Figure 6.1 shows how much space consumes each table. MaxDB allocates memory for all attributes of tuple. If a value of some attribute is null, database allocates a free space for this value, which can be inserted in the future. This type of storing data brings a big difference in memory consumption between MaxDB and HDB database.

In this chart, there is also very a noticeable difference between row and column layout of data in memory. With dictionary encoding we can save more space in memory thanks to lot of same values in attributes `timezoneid` and `countrycode`. In table `ZMUCITY` I achieved compression rate 5, but I have to mention one more thing. HANA database uses non-volatile memory to store data and it is important to have another disk to store backup of database. Unfortunately, I could not get information about space needed for backups in HANA database, but due to decreasing prices of hard disks this information loses its importance.

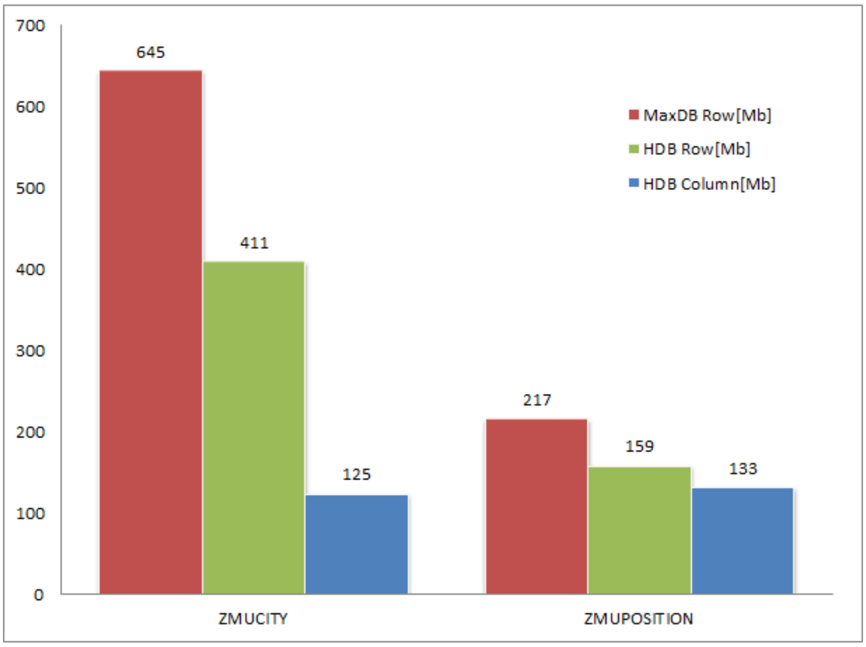


Figure 6.1: Size of tables.

6.2 Inserting into tables

Insertion into database is really time consuming process. The best potency for insertions has HDB with row storage layout. Values in chart shown in Figure 6.2 are calculated using average value of all measurements. Insertion into column store is two times slower than row store insertion in this case, but it is still very good performance, if we consider all the operations that are carried out with dictionary encoding. This measurement does not include the time that is needed for merge process, but this process runs in background and all queries can be executed.

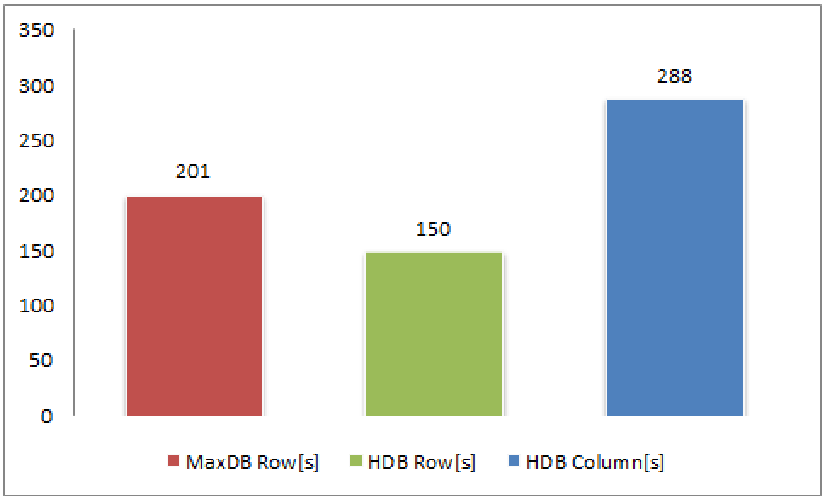


Figure 6.2: Insert into database.

6.3 Selection tests

The main part of this thesis is the test result. For testing, I prepared four queries focused to tuple reconstruction and four queries focused to column operations with aggregate functions. For these tests, I expect that queries focused on row operations will be faster with row oriented layout and column operations will be much faster with column oriented layout. My next expectation is, that HDB will be always faster than MaxDB because of using memory with shorter access time. This comparison should show the difference between database bottlenecks that I discuss in chapter 6.4. In sections below, I introduce each test with its SELECT statement. I also discuss the results of test shown in chart and differences in execution plans of row and column oriented layout in HDB. Execution plan of each test is included at the end of this chapter for better clarity of document.

6.3.1 Test 1

```
SELECT *
   FROM zmucity
```

This command select everything from table with city informations.

First test is focused on reconstruction of all tables selecting all attributes tuple per tuple. From Figure 6.3 is obvious, that best storage for table reconstruction is definitely HDB with row oriented layout. The same layout of data on MaxDB and HDB, but different type of storage caused, that the second one is four times faster. Execution plan in Figure 6.13 shows the late materialization as the main reason, why the column oriented layout of data in memory is slower than row oriented one, because data must be encoded.

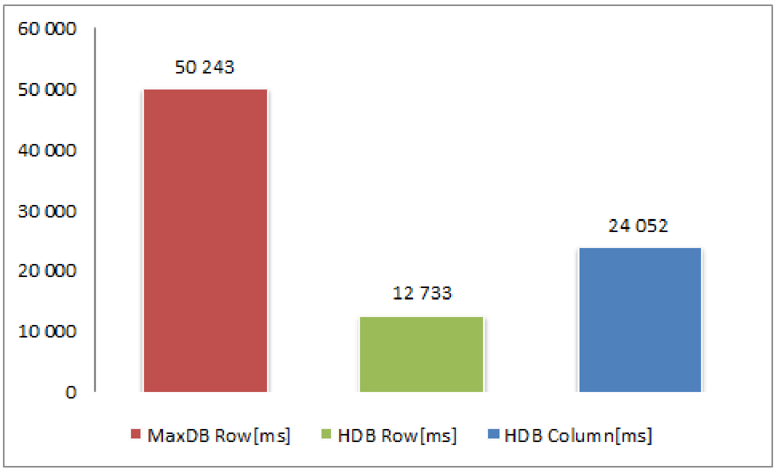


Figure 6.3: Results of Test 1.

6.3.2 Test 2

```
SELECT *
   FROM zmucity JOIN zmuposition
   ON zmucity.id = zmuposition.id
```

This command join city informations with position informations about cities on their ID.

This test is very similar to the first one, but it contains `JOIN` condition to join two tables. Execution times shown in Figure 6.4 are more than two times slower than times in Test 1, because both tables has 5,000,000 tuples and the database selects them all. What is really interesting, is the execution plan in Figure 6.14. In row store layout, table `ZMUCITY` is scanned, then joined with `ZMUPOSITION` using index join. In contrast, optimizer in column store layout first scans both tables and then joins them together.

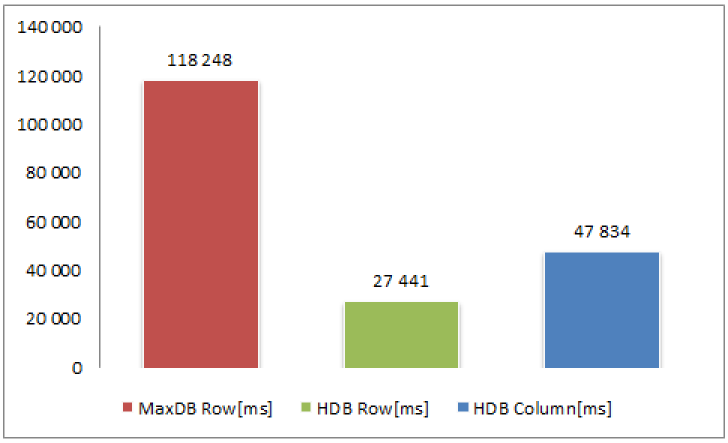


Figure 6.4: Results of Test 2.

6.3.3 Test 3

```
SELECT *
   FROM zmucity
   WHERE countrycode = 'CA' OR countrycode = 'CO'
```

Select only cities with country code CA for Canada and CO for Colombia.

In this test, simple WHERE condition is added and column store is becoming faster. Results can be seen in Figure 6.5. Execution plans in Figure 6.15 are very similar. To answer the question, why reconstruction part of table defined with condition is faster in column layout, I have to go deep in the dictionary encoding. In dictionary encoded column, the dictionary is scanned for value defined in WHERE condition and valueID is found for these values. Then the attribute vector is scanned for valueID comparing integers. The result of this operation is list of positionIDs and late materialization is the last step. In contrast in non encoded column, each row must be compared string by string and this operation is more expensive than comparing integers. There would be really interesting to see, how much time consumes filtering in each data layouts, because from previous tests we know, that materialization is prolongs significantly the run time.

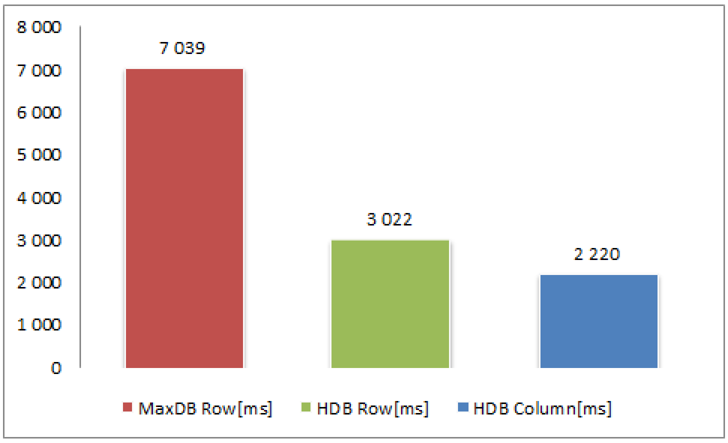


Figure 6.5: Results of Test 3.

6.3.4 Test 4

```
SELECT zmucity.id, zmucity.name, zmuposition.latitude, zmuposition.longitude
   FROM zmucity JOIN zmuposition
   ON zmucity.id = zmuposition.id
   WHERE latitude BETWEEN 20 AND 50 AND
   longitude BETWEEN 30 AND 60
```

Select city ID, name, latitude and longitude only from cities where latitude is between 20 and 50 and longitude is betweeb 30 and 60.

Result of this test in Figure 6.6 looks very similar to test 3, but there is one question, you might have. Why is column layout faster even if in `WHERE` condition we compare integers with integers and string comparison is not slowing down selection anymore?

The answer can be found in chapter 3.2. In column store, data is stored attribute-wise so during the scan, the column memory is accessed byte per byte, but in row store, data is stored tuple-wise and we have to read few bytes and then skip to read next required bytes.

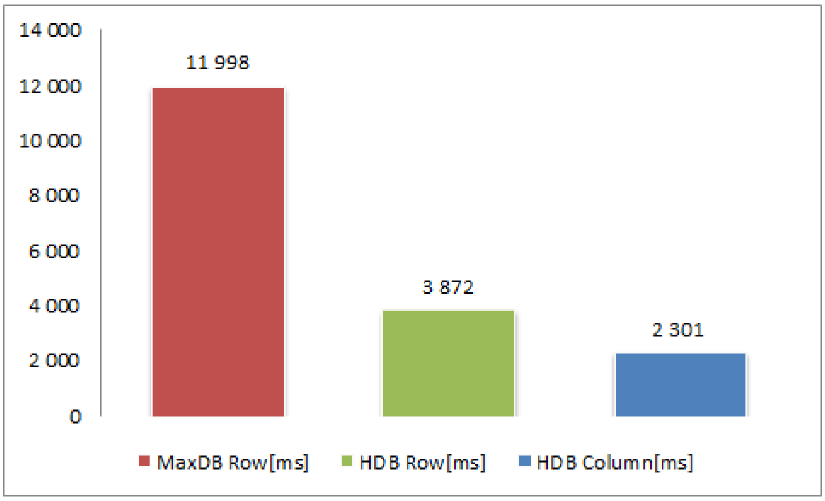


Figure 6.6: Results of Test 4.

6.3.5 Test 5

```
SELECT SUM( latitude )
   FROM zmuposition
```

Select summary of latitude value. This command is nonsense in real life, but it tests single column operation.

Next four tests are focused on column operations. This one is a simple `SELECT` using aggregate function to compute summary of one attribute. As shown in Figure 6.7, for this type of query is HANA database with columnar layout the most suitable. MaxDB is more than 60 times slower. Materialization is not used, because this type of query returns only one number.

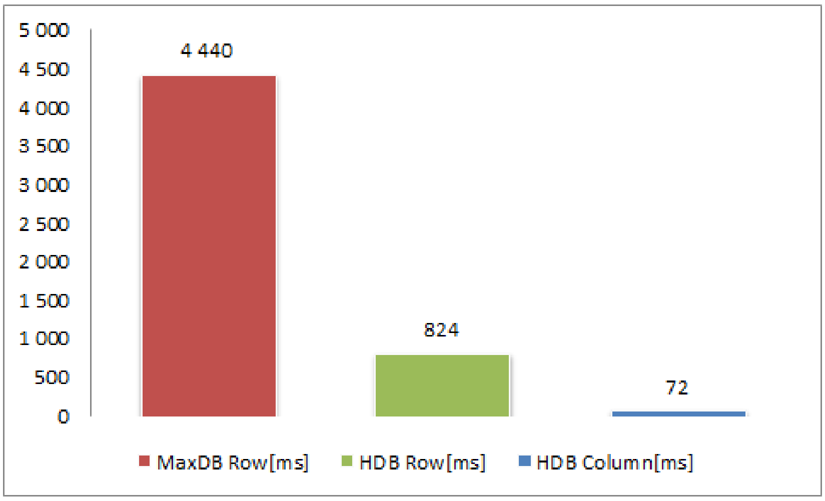


Figure 6.7: Results of Test 5.

6.3.6 Test 6

```
SELECT AVG( population )
   FROM zmucity
   GROUP BY population
   HAVING population > 2000
```

This test results in Figure 6.8 shows, that with `GROUP BY` and `HAVING` condition, the difference between row and column layout is not significant. If you follow execution plan in Figure 6.18, at first, table is filtered by condition in clause `HAVING`, then rows are grouped, aggregated and prepared for output. How the aggregation works is explained in chapter 4.4.

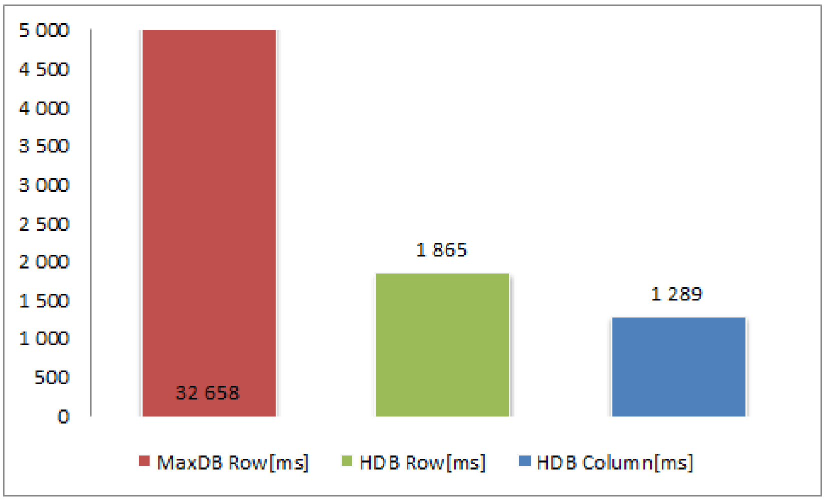


Figure 6.8: Results of Test 6.

6.3.7 Test 7

```
SELECT SUM( gmt )
   FROM zmucity JOIN zmutimezones
   ON zmucity.timezoneid = zmutimezones.timezoneid
   WHERE population > 3000 AND
   zmutimezones.countrycode = 'CA' AND
   zmutimezones.countrycode = 'AZ'
```

This command returns summary of gmt for cities, where population is over 3000 and country code is CA (Canada) and AZ (Azerbaijan).

This test is focused to join table with almost 500 tuples with table with 5,000,000 tuples. In contrast to test 8 (see below), this join has big difference on MaxDB, but the difference on HANA database is not too big as shown in Figure 6.9. This test also confirms my hypothesis, that column store is faster than row store.

Interesting in this test is execution plan in Figure 6.19. If you look into the detail, the ways how to execute one SELECT statement could be very different depending on the storage technique. For row store, smaller table is scanned first with filter, then bigger table is scanned, hash join (explained in chapter 4.3) is applied with additional filtering and the result is aggregated. For column store, tables are scanned in the same order, then joined and before aggregation additional filtering is performed. In my opinion, it would be faster to apply condition POPULATION > 3000 during table ZMUCITY scan in second step and then join filtered tables.

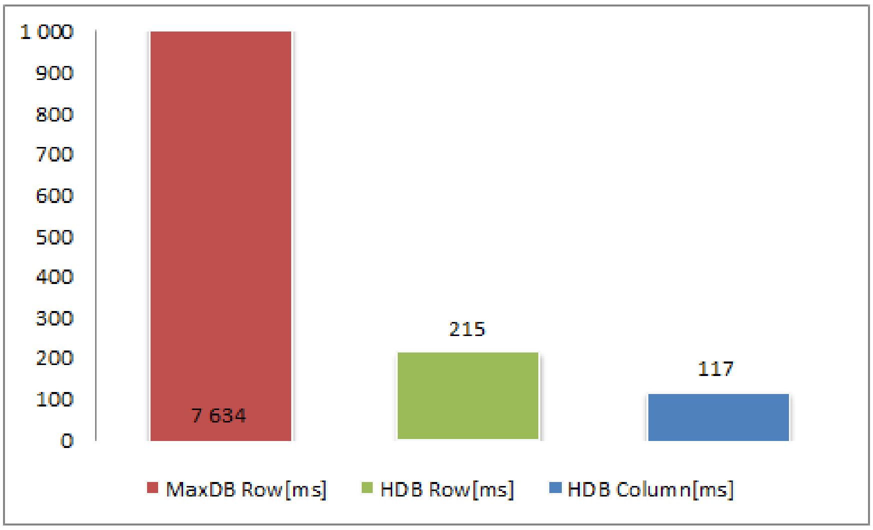


Figure 6.9: Results of Test 7.

6.3.8 Test 8

```
SELECT AVG( latitude )
   FROM zmucity JOIN zmuposition
   ON zmucity.id = zmuposition.id
   WHERE population > 3000 AND
   countrycode = ’CA’ AND
   countrycode = ’AZ’
```

In this case we ask for average latitude of cities with population over 3000 and country code CA (Canada) and AZ (Azerbaijan).

In last test, I reached the biggest gap between MaxDB and HDB. This gap is obvious in Figure 6.10 Execution time on HDB columns store is more than 280 times shorter then on MaxDB. This query joins tables, both with 5,000,000 tuples, and filters values with three conditions. From this subset, average value is calculated.

Also in this test, the execution plans are very different. In a row store layout, table ZMUCITY is scanned and filtered by conditions, then joined with table ZMUPOSITION and average value is calculated at the end. In column store layout, tables change order, so ZMUPOSITION is scanned first, then ZMUCITY is scanned and filtered, inner join is applied and aggregation runs also at the end. In this case, first two steps can be executed in parallel, but I think, that this is future for HDB, or it is already running in parallel, but unfortunately, the execution plan does not contain this information.

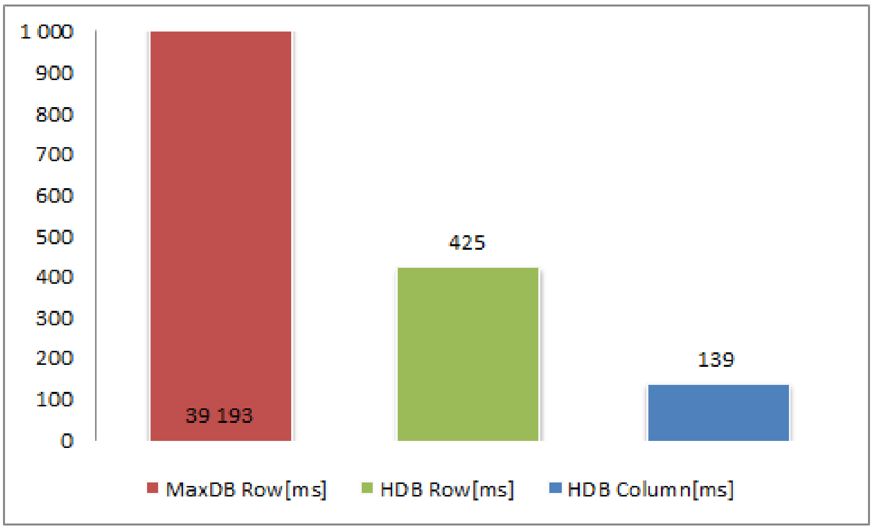


Figure 6.10: Results of Test 8.

6.4 Tests summary

Summing up my measurements, it is really obvious, that MaxDB cannot compete with HANA. Migration of storage into main memory provides much faster database, enabling us to have one source of data for OLTP and OLAP operations and to execute them on the fly. This was the main reason, why prof. Plattner started to think about a new concept of database.

On the other hand, the non-volatile storages are still needed and hard disks are gradually replaced by solid state disks with faster access. This type of storage is used for database backups and logging of recent changes. Unfortunately, it is not in my competence to test how reliable is process of database recovery after fail, or after blackout which is described in detail in chapter 3.6.2

In section I said, that only with change hard disk with main memory we can achieve 100,000 times faster access into database and my tests are not fulfill this. The main reason is, that also MaxDB uses main memory to cache data. So, data is loaded into cache with first test execution and then processed through the cache.

The last thing I would like to mention, is that it is very difficult to decide, which type of layout of data in main memory is better. If I sum up results of my tests, column store takes less space in memory and most of my tests run faster using the column store. Table definition in ABAP workbench offers developers both options, so the development team will always determine which storage type will provide better table usage performance. The default storage type for tables in HANA database is column store. I found official recommendation for SAP developers, stating which layout should be used in which cases [3].

Reasons why a table should be in row store:

- Very huge OLTP load in the table (huge rate of single updates / inserts / deletes).

Reasons why a table must be in column store:

- Tables with the new data types TEXT, SHORTTEXT, ALPHANUM must be in column store because this functionality is not (yet) available in row store.
- Tables with fulltext index.
- Tables with many rows (more than multiple 100,000) because of better compression in a column store.
- Table is used in analytical context and contains business relevant data.

6.5 Performance stability

At the end of this chapter, I would like to mention one interesting thing. As I wrote at the beginning of this chapter, I performed ten measurements and then I calculated average value. In chart shown in Figure 6.11, individual values can be seen from insertion into database and in chart shown in Figure 6.12 values of Test 4.

It is really obvious, that the lines of MaxDB (red) are not direct, but curved and for HDB with column layout (blue) of data the line is curved too. The reason of instable MaxDB is its storage. This database uses hard disk to store data and duration of the movement of its mechanical parts cannot be calculated. In contrast, the main memory scan of HANA database is undoubtedly deterministic and always with same duration. But, why the insertion into HDBs column store is also not stable when compared to HDBs row store? I hope, that the answer is already known from section 6.2, but if not, the answer is dictionary encoding.

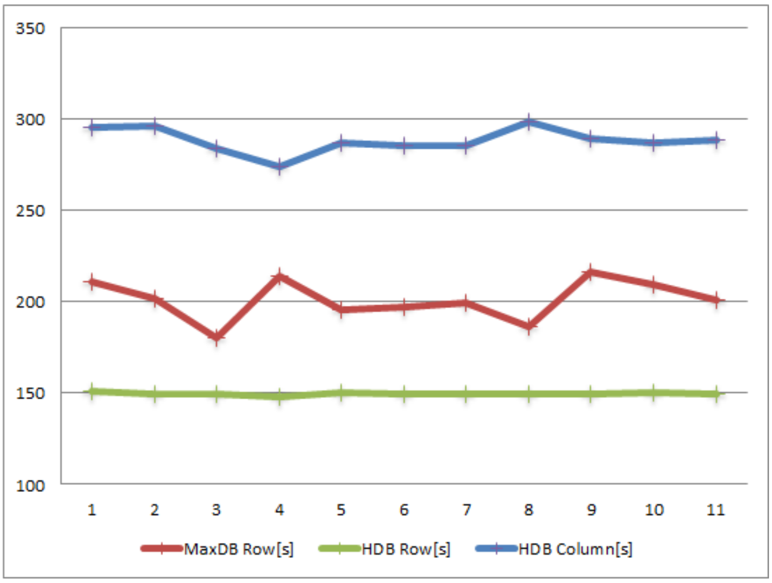


Figure 6.11: INSERT execution.

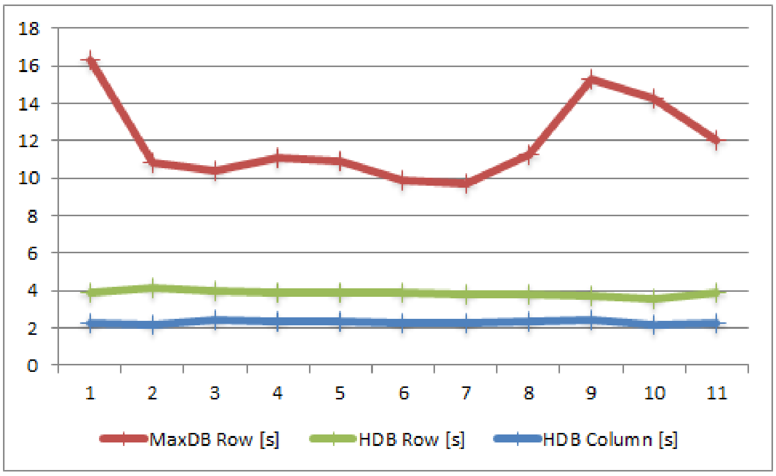
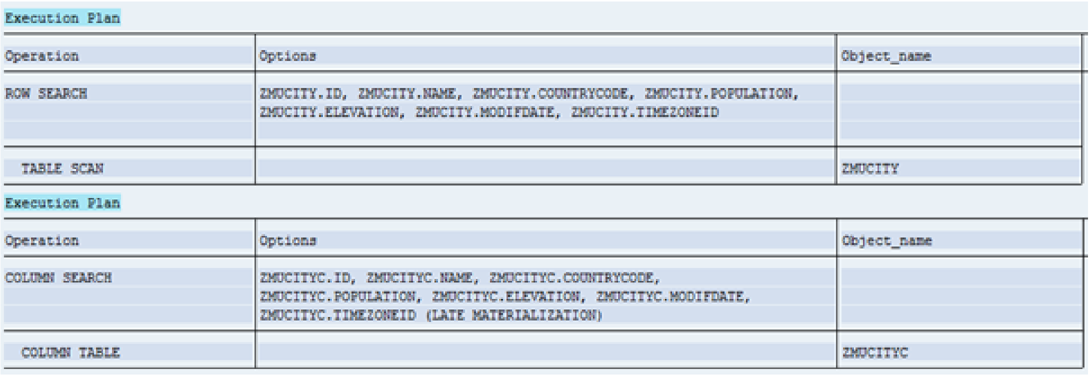


Figure 6.12: SELECT execution.

6.6 Execution plans

This section contains execution test plans for each test. In each figure, first plan is for row oriented tables and second one is for column oriented tables stored in HDB.



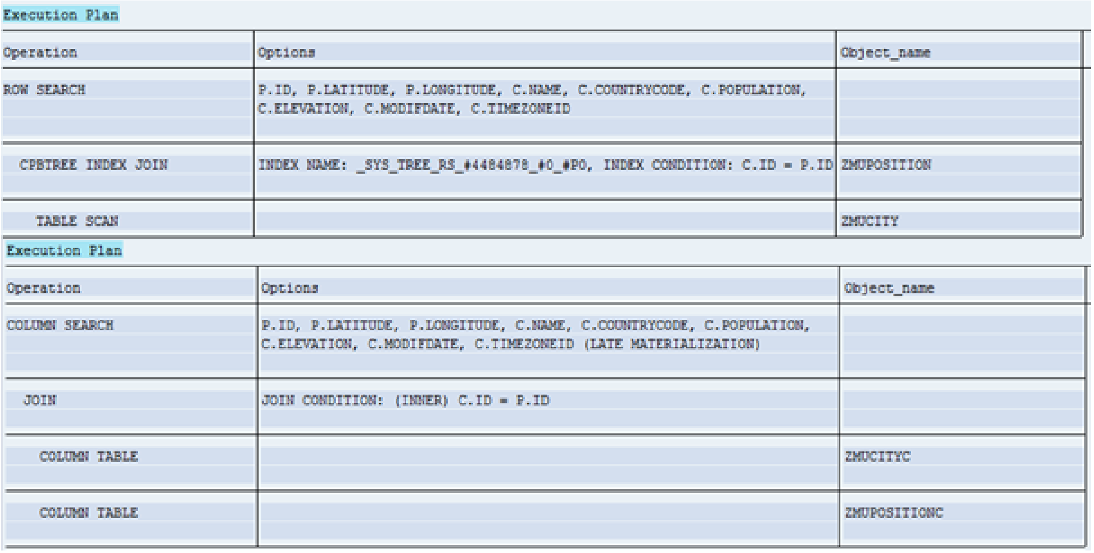
Execution Plan

Operation	Options	Object_name
ROW SEARCH	ZMUCITY.ID, ZMUCITY.NAME, ZMUCITY.COUNTRYCODE, ZMUCITY.POPULATION, ZMUCITY.ELEVATION, ZMUCITY.MODIFDATE, ZMUCITY.TIMEZONEID	
TABLE SCAN		ZMUCITY

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	ZMUCITYC.ID, ZMUCITYC.NAME, ZMUCITYC.COUNTRYCODE, ZMUCITYC.POPULATION, ZMUCITYC.ELEVATION, ZMUCITYC.MODIFDATE, ZMUCITYC.TIMEZONEID (LATE MATERIALIZATION)	
COLUMN TABLE		ZMUCITYC

Figure 6.13: Execution plans of Test 1.



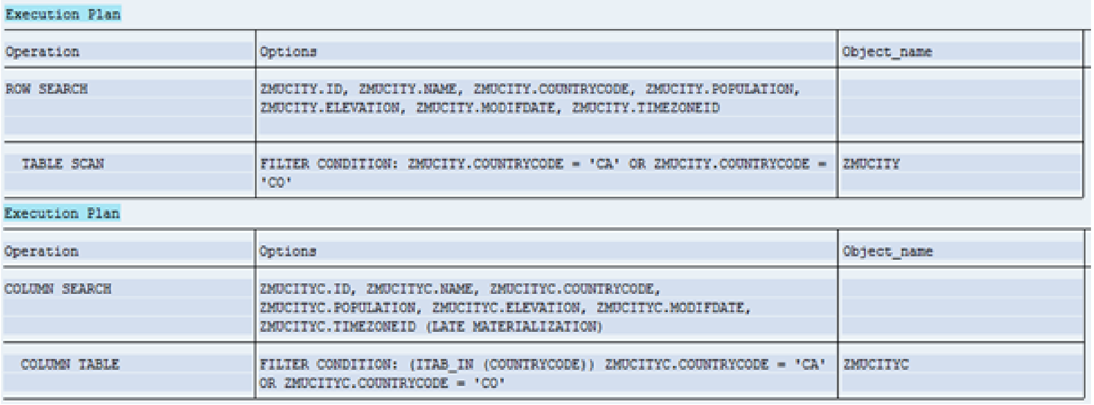
Execution Plan

Operation	Options	Object_name
ROW SEARCH	P.ID, P.LATITUDE, P.LONGITUDE, C.NAME, C.COUNTRYCODE, C.POPULATION, C.ELEVATION, C.MODIFDATE, C.TIMEZONEID	
CPBTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#4484878_#0_#P0, INDEX CONDITION: C.ID = P.ID	ZMUPOSITION
TABLE SCAN		ZMUCITY

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	P.ID, P.LATITUDE, P.LONGITUDE, C.NAME, C.COUNTRYCODE, C.POPULATION, C.ELEVATION, C.MODIFDATE, C.TIMEZONEID (LATE MATERIALIZATION)	
JOIN	JOIN CONDITION: (INNER) C.ID = P.ID	
COLUMN TABLE		ZMUCITYC
COLUMN TABLE		ZMUPOSITIONC

Figure 6.14: Execution plans of Test 2.



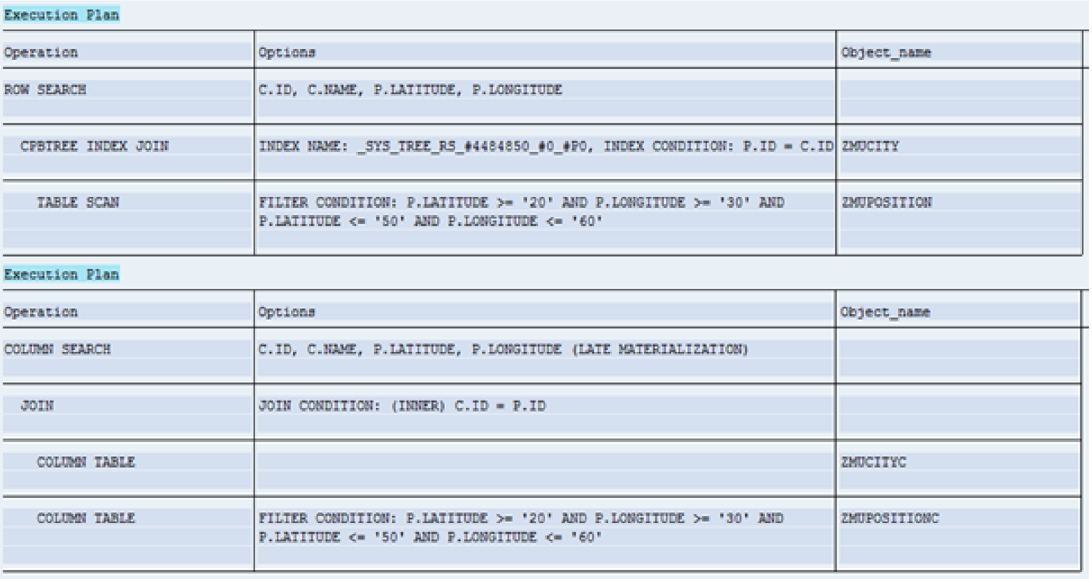
Execution Plan

Operation	Options	Object_name
ROW SEARCH	ZMUCITY.ID, ZMUCITY.NAME, ZMUCITY.COUNTRYCODE, ZMUCITY.POPULATION, ZMUCITY.ELEVATION, ZMUCITY.MODIFDATE, ZMUCITY.TIMEZONEID	
TABLE SCAN	FILTER CONDITION: ZMUCITY.COUNTRYCODE = 'CA' OR ZMUCITY.COUNTRYCODE = 'CO'	ZMUCITY

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	ZMUCITYC.ID, ZMUCITYC.NAME, ZMUCITYC.COUNTRYCODE, ZMUCITYC.POPULATION, ZMUCITYC.ELEVATION, ZMUCITYC.MODIFDATE, ZMUCITYC.TIMEZONEID (LATE MATERIALIZATION)	
COLUMN TABLE	FILTER CONDITION: (ITAB_IN (COUNTRYCODE)) ZMUCITYC.COUNTRYCODE = 'CA' OR ZMUCITYC.COUNTRYCODE = 'CO'	ZMUCITYC

Figure 6.15: Execution plans of Test 3.



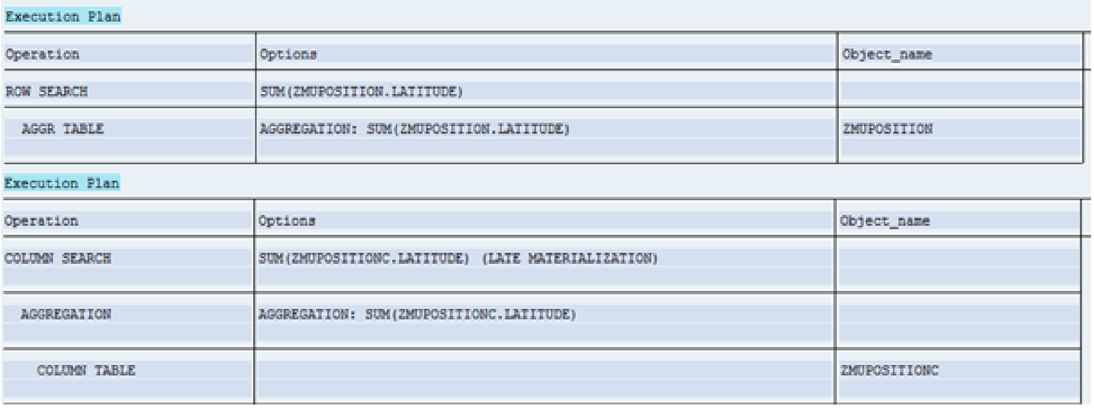
Execution Plan

Operation	Options	Object_name
ROW SEARCH	C.ID, C.NAME, P.LATITUDE, P.LONGITUDE	
CPBTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#4484850_#0_#P0, INDEX CONDITION: P.ID = C.ID	ZMUCITY
TABLE SCAN	FILTER CONDITION: P.LATITUDE >= '20' AND P.LONGITUDE >= '30' AND P.LATITUDE <= '50' AND P.LONGITUDE <= '60'	ZMUPOSITION

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	C.ID, C.NAME, P.LATITUDE, P.LONGITUDE (LATE MATERIALIZATION)	
JOIN	JOIN CONDITION: (INNER) C.ID = P.ID	
COLUMN TABLE		ZMUCITYC
COLUMN TABLE	FILTER CONDITION: P.LATITUDE >= '20' AND P.LONGITUDE >= '30' AND P.LATITUDE <= '50' AND P.LONGITUDE <= '60'	ZMUPOSITIONC

Figure 6.16: Execution plans of Test 4.



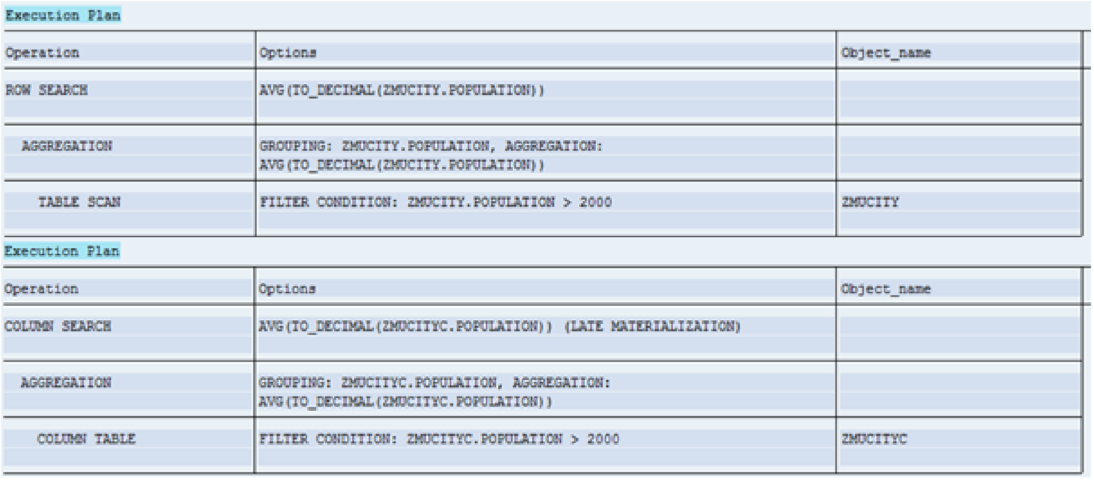
Execution Plan

Operation	Options	Object_name
ROW SEARCH	SUM(ZMUPOSITION.LATITUDE)	
AGGR TABLE	AGGREGATION: SUM(ZMUPOSITION.LATITUDE)	ZMUPOSITION

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	SUM(ZMUPOSITIONC.LATITUDE) (LATE MATERIALIZATION)	
AGGREGATION	AGGREGATION: SUM(ZMUPOSITIONC.LATITUDE)	
COLUMN TABLE		ZMUPOSITIONC

Figure 6.17: Execution plans of Test 5.



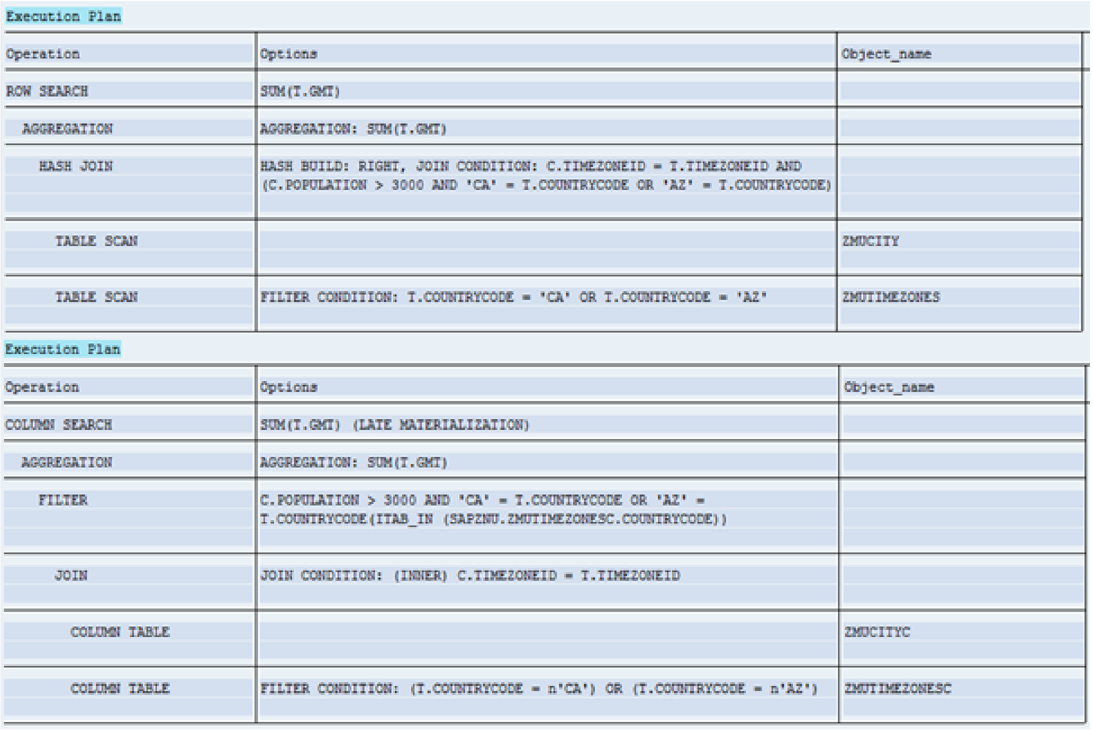
Execution Plan

Operation	Options	Object_name
ROW SEARCH	AVG(TO_DECIMAL(ZMUCITY.POPULATION))	
AGGREGATION	GROUPING: ZMUCITY.POPULATION, AGGREGATION: AVG(TO_DECIMAL(ZMUCITY.POPULATION))	
TABLE SCAN	FILTER CONDITION: ZMUCITY.POPULATION > 2000	ZMUCITY

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	AVG(TO_DECIMAL(ZMUCITYC.POPULATION)) (LATE MATERIALIZATION)	
AGGREGATION	GROUPING: ZMUCITYC.POPULATION, AGGREGATION: AVG(TO_DECIMAL(ZMUCITYC.POPULATION))	
COLUMN TABLE	FILTER CONDITION: ZMUCITYC.POPULATION > 2000	ZMUCITYC

Figure 6.18: Execution plans of Test 6.



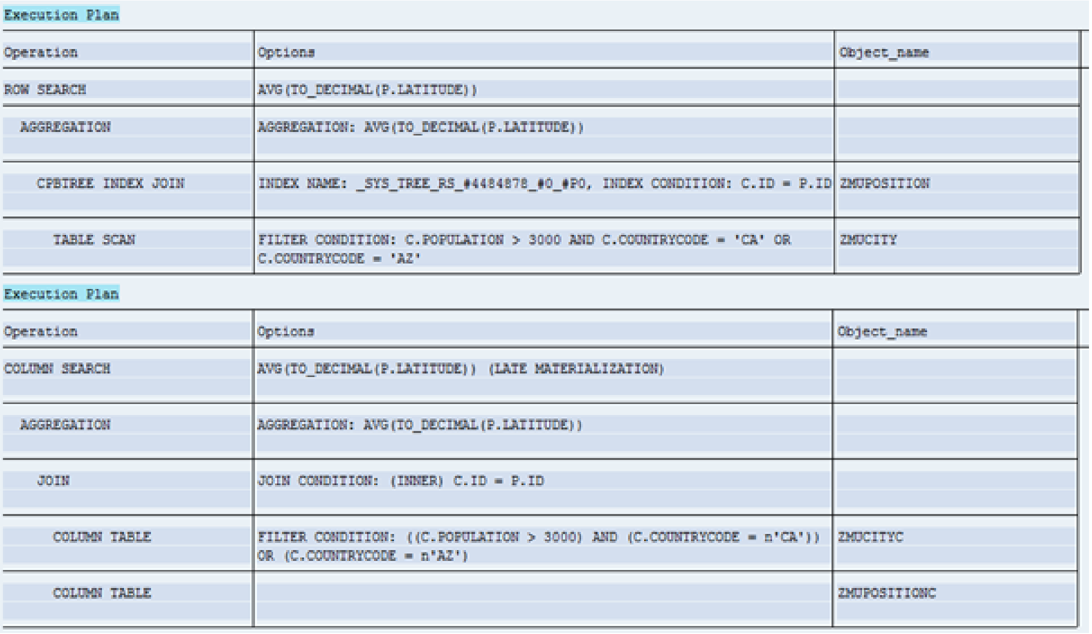
Execution Plan

Operation	Options	Object_name
ROW SEARCH	SUM(T.GMT)	
AGGREGATION	AGGREGATION: SUM(T.GMT)	
HASH JOIN	HASH BUILD: RIGHT, JOIN CONDITION: C.TIMEZONEID = T.TIMEZONEID AND (C.POPULATION > 3000 AND 'CA' = T.COUNTRYCODE OR 'AZ' = T.COUNTRYCODE)	
TABLE SCAN		ZMUCITY
TABLE SCAN	FILTER CONDITION: T.COUNTRYCODE = 'CA' OR T.COUNTRYCODE = 'AZ'	ZMUTIMEZONES

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	SUM(T.GMT) (LATE MATERIALIZATION)	
AGGREGATION	AGGREGATION: SUM(T.GMT)	
FILTER	C.POPULATION > 3000 AND 'CA' = T.COUNTRYCODE OR 'AZ' = T.COUNTRYCODE(ITAB_IN (SAPZNU.ZMUTIMEZONESC.COUNTRYCODE))	
JOIN	JOIN CONDITION: (INNER) C.TIMEZONEID = T.TIMEZONEID	
COLUMN TABLE		ZMUCITYC
COLUMN TABLE	FILTER CONDITION: (T.COUNTRYCODE = n'CA') OR (T.COUNTRYCODE = n'AZ')	ZMUTIMEZONESC

Figure 6.19: Execution plans of Test 7.



Execution Plan

Operation	Options	Object_name
ROW SEARCH	AVG(TO_DECIMAL(P.LATITUDE))	
AGGREGATION	AGGREGATION: AVG(TO_DECIMAL(P.LATITUDE))	
CPBTREE INDEX JOIN	INDEX NAME: _SYS_TREE_RS_#4484878_#0_#P0, INDEX CONDITION: C.ID = P.ID	ZMUPOSITION
TABLE SCAN	FILTER CONDITION: C.POPULATION > 3000 AND C.COUNTRYCODE = 'CA' OR C.COUNTRYCODE = 'AZ'	ZMUCITY

Execution Plan

Operation	Options	Object_name
COLUMN SEARCH	AVG(TO_DECIMAL(P.LATITUDE)) (LATE MATERIALIZATION)	
AGGREGATION	AGGREGATION: AVG(TO_DECIMAL(P.LATITUDE))	
JOIN	JOIN CONDITION: (INNER) C.ID = P.ID	
COLUMN TABLE	FILTER CONDITION: ((C.POPULATION > 3000) AND (C.COUNTRYCODE = n'CA')) OR (C.COUNTRYCODE = n'AZ')	ZMUCITYC
COLUMN TABLE		ZMUPOSITIONC

Figure 6.20: Execution plans of Test 8.

Chapter 7

Conclusion

SAP HANA is undoubtedly a new technology that will be used in the future mainly due to the spread between the customers of SAP. In my testing, I found that the difference between the performance of SAP HANA and MaxDB is very evident, but it is very good for a potential customer to consider, whether it is really necessary for his/her need to have such a powerful database. The SAP developers have a difficult task to optimize their modular system to run on HANA, because a poorly written code can increase the execution time. To exploit the full potential, it is also imperative that developers understand the basic principles of in memory database and utilize the gained knowledge during the development process.

I would like to apply the knowledge gained in this thesis in my master thesis. My goal will be to become familiar with the development tool SAP HANA STUDIO, based on Eclipse and with SAPUI5 library. This also requires to get knowledge on OData service, which provides data from database. To get practical experience with these tools, I would like to create an application, which needs to run with an in-memory database.

Bibliography

[1] Saplink. [online], 2010. DostupnĂŠ z: https://www.code.google.com/p/saplink/. [cit. 06.03.2014].

[2] abap2xlsx. [online], 2012. DostupnĂŠ z: http://ivanfemia.github.io/abap2xlsx/. [cit. 08.04.2014].

[3] Row store vs. column store. [online], 2013. DostupnĂŠ z: https://wiki.wdf.sap.corp/wiki/display/ngdb/Row+Store+vs.+Column+Store. [cit. 09.04.2014].

[4] Bjarne BERG and Silvia PENNY. *SAP HANA: An Introduction.* SAP PRESS, 2012. ISBN 978-1-59229-434-3.

[5] Andre BÖGELSACK, Stephan GRADL, Manuel MAYER, and Helmut KRCMAR. *SAP MaxDB Administration.* Bonn: Galileo Press, 2009. ISBN 978-1-59229-299-8.

[6] Franz FÄRBER, Sang Kyun CHA, Jürgen PRIMSCH, Christof BORNHÖVD, Stefan SIGG, and Wolfgang LEHNER. Sap hana database. *ACM SIGMOD Record*, 40:45–51, 2012. DostupnĂŠ z: http://dl.acm.org/citation.cfm?doid=2094114.2094126.

[7] Michael KIFER, Arthur BERNSTEIN, and Philip M. LEWIS. *Database Systems.* Boston: Addison-Wesley, 2005. ISBN 03-212-6845-8.

[8] Hasso PLATTNER. *A Course in In-Memory Data Management.* Springer, 2013. ISBN 978-3-642-36523-2.

[9] James WOOD. *ABAP cookbook: programming recipes for everyday solutions.* Boston: Galileo Press, 2010. ISBN 978-1-59229-326-1.

Appendix A

Content of CD

- source - .NUGG files, to install application (reports, tables, table types) to system
  - ZKU_RFCs - source codes of each RFC functions have to be installed manually
- thesis - source files for LaTeXof this thesis
  - img - figures