# Row-Store / Column-Store / Hybrid-Store

Kevin Sterjo

December 11, 2017

**Abstract**

Three of the most widely used main memory database system layouts available today are row store, column store and hybrid store. In this paper, their similarities and differences regarding their layout and the way each of them handle data are presented based on available literature and two simple implementations. The results of the implementations, not always matching the theoretical findings, present a simple understanding of how different approaches and additional techniques may change the results drastically.

## 1 Introduction

A Main Memory Database System (MMDB) is a database management system (DBMS) that stores the entire database content in the physical main memory, opposite to conventional DBMS, which are designed for the use of disk storage [1, 11]. The main advantage of the MMDB over normal DBMS is a significantly lower I/O cost, since the data lives permanently in the memory, which translates to very high-speed access [1].

Row store, column store and hybrid store are kinds of database layouts. In my paper I will focus on the main memory versions of them. In the conventional row store each entity is assigned a dedicated row [1, 10]. Column store stores tables of tuple attributes contiguously [2, 7]. Hybrid store is a combination of both of these layouts, storing insert and update intense data in row stores and analytical and historical data in column stores [1, 12].

In the upcoming sections, I'll introduce three different storing techniques, namely row-store, column-store and hybrid-store and compare the advantages and disadvantages of each, their main purpose and where they find use based on the requirements of different services. I then introduce a simple C++ implementation of each type of layout and show how the advantages of one over the other in terms of time needed to compute specific operations over different table sizes.

## 2 Existing storing Methods for In-Memory Databases

In this section, I will discuss more in detail the characteristics, advantages and disadvantages of the three different layouts, namely row store, column store and hybrid and provide the most common use cases for each.

### 2.1 Row Store

Row oriented database systems (also known as row stores) represent the traditional way the data is physically stored. As the name suggests, the database stores all the attributes values of each tuple subsequently. Hence, when one needs to access attributes of a tuple, regardless of how many are relevant to the operation, each tuple attribute will still be accessed. The advantages of this kind of storage can be seen when multiple attributes of a single row need to be accessed. When, in the opposite scenario, only single attribute values need to be accessed, this storage layout can be at a disadvantage because of the costly unnecessary usage of cache storage [1, 10]. In row store, each tuple attribute will be transferred to the Level2 and Level1 caches, alongside the necessary value. The summation of the values in a single column of a table is considered:

```
select sum(A)
from R
```

The layout of the stored attributes in the main memory alongside with the transfers from the main memory to the cache lines needed for the operations considered are displayed in Figure 1.

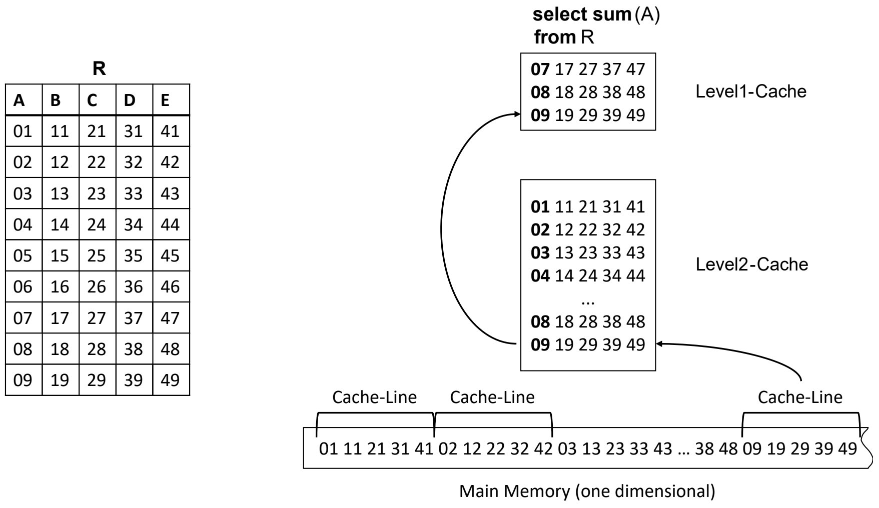


Figure 1: Logical Row Store. Source: own representation based on [1]

Using row store and considering the example of the sum of the first column of the table, it is clear that this kind of storage transfers roughly four times more information than the relevant values needed for the operation. In this example the main memory is accessed 9 times using row store. This is only a simplified example of memory access and cache sizes. In reality, cache lines usually have a size of 64 or 128 Bytes [1].

Since row store handles data “horizontally” (accessing rows of a table resembles reading the table horizontally), they are more suited for updates and insertions. In row stores, the whole row is written in a single operation [10]. This makes row stores more preferable for OLTP-oriented databases, since OLTP workloads tend to be more loaded with interactive transactions, like retrieving every attribute from a single entity or adding entities to the table [6].

Some of the most well-known databases that store the data using this technique are Oracle, IBM DB2, Microsoft SQL Server and MySQL.

## 2.2 Column Store

Column-oriented database systems (column-stores) have only recently gathered the attention of many database architects even though their origins can be dated back to the 1970s and their advantages over typical relational databases being documented since the late 1980s [7].

A column-store database layout differs from the traditional row-store in the very core of how data is stored in the memory. In a column-store, attributes depicted by column are stored contiguously as depicted in Figure 2 [7].

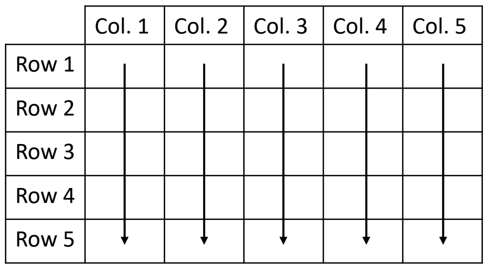


Figure 2: Structure of a Column oriented database. Source: own representation based on [7]

The same example used in row store, where the summation of one column is required, can be implemented here as well. Using the same values as in the first example, but this time "splitting" the table into smaller columnar tables for each attribute, it is clear that these columns are stored contiguously in the main memory. Furthermore, given the same query and the same cache line size as before, Figure 3 shows how these values are accessed and copied into the Level2 and Level1 caches. The cache lines in this scenario are way less overloaded with redundant information as in the row store layout. It can also be observed that this time the main memory is only accessed two times for the cache line transfers compared to nine costly transfers in row store [1]. The physical storage of a column store depends on the intended use of the database.

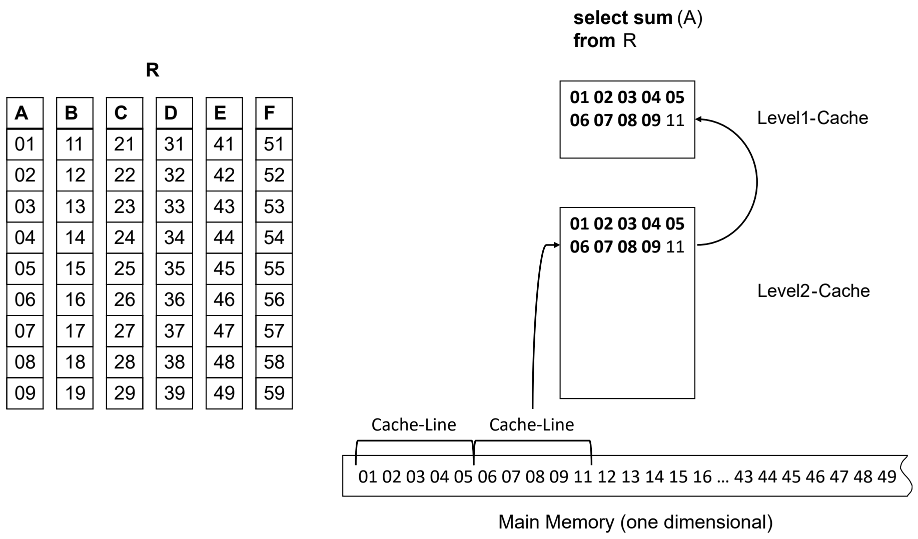


Figure 3: Logical Column Store. Source: own representation based on [1]

Column stores have lately been subject to a vast interest in the design of read-optimized database systems. While, as mentioned above, row stores seem to be better suited at write-oriented cases in which whole tuples are accessed, column stores have shown to be more efficient at handling read-oriented cases when specific subsets of a table containing a large number of tuples are targeted [6]. Recent research has proven that column stores perform an order of magnitude better than row stores on analytical workloads such as those in data warehouses, decision support and business intelligence applications [2, 4, 7]. Because of this significant advantage, column stores have been successfully implemented in OLAP-oriented databases [9]. However while column store is more performant regarding reading operations, it suffers in performance in writing ones. The reason write operations and tuple construction operations are problematic is that each entity has to be split into each of its attributes and consequently they have to be written separately [3]. The difference in how the data is accessed for different processes depending on the layout of the database can be seen in Figure 4.

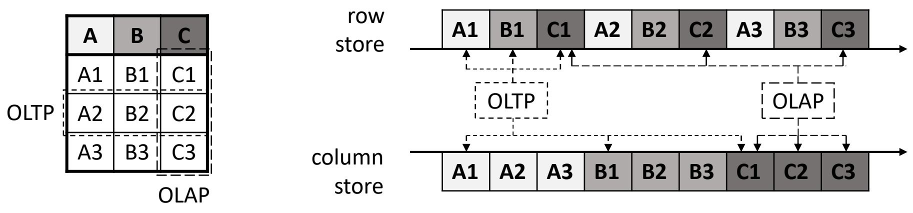
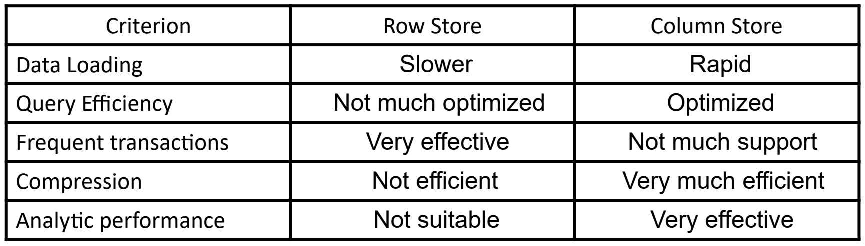


Figure 4: Memory alignment and resulting access patterns for row and column stores. Source: own representation based on [8]

A small summary of the advantages and disadvantages of column store over row store and vice versa can be seen in Figure 5.



| Criterion | Row Store | Column Store |
|---|---|---|
| Data Loading | Slower | Rapid |
| Query Efficiency | Not much optimized | Optimized |
| Frequent transactions | Very effective | Not much support |
| Compression | Not efficient | Very much efficient |
| Analytic performance | Not suitable | Very effective |

Figure 5: Comparative analysis of row and column store. Source: own representation based on [7]

Some of the most known commercially available column store databases are MonetDB, C-Store, Sybase IQ the SAP BI accelerator, and Kickfire.

## 2.3 Hybrid Store

Many businesses require that their databases are able to handle OLTP transactions and OLAP processes without prioritizing one over the other. In such databases it can be difficult to choose only one layout, since each stores data differently and profits from different advantages when compared to the other. With this challenge in mind, database architects had to develop a new type of layout. This marked the inception of the idea of hybrid store [1].

In hybrid store, a combination of both techniques discussed above is used. Some of the attributes are grouped together to form a “mini-table” which will act as the row-store component. The remaining attributes are stored individually in single columnar tables and they act as the column-store component. Since the properties the layouts bring are almost mutually exclusive, the most meaningful way of separating the attributes is by its use case [12]. If it is already known which components of the entity will be needed for transaction processing, where the whole group of attributes for the operation is needed or where additional data is to be written or new data is to be entered on these attributes, the row oriented database would be preferred [6]. Using the same approach, it is clear that the attributes which will mostly be used for analytical processes would need to be stored in the column store component [5].

## 3 Implementation: Row-Store vs. Column-Store vs. Hybrid-Store

Based on the various database storage technologies collected through the previously conducted literature review, a simple simulation showing these approaches would help further understand their differences. I implemented such an example using C++. More importantly, I used only the basic libraries provided by the C++ 98 iteration of the language to ensure getting closer to an actual low-level implementation of a database.

### 3.1 Initial Effort

Since C++ is an object-oriented language, the initial approach consisted in the usage of such objects as storage containers for the columns or rows. To identify the appropriate object types required for a similar storage structure to that of a database, it was also necessary to look at the memory allocation C++ internally uses to store these objects. The basic C++ objects **"vector"** (an expandable array of same-type objects) and **"struct"** (custom-defined structure consisting of nested objects) were ideal for this simple implementation because C++ stores these objects sequentially in memory and avoids the usage of pointers that may jump to different memory locations.

The sample table used for the simulation was provided in the form of a comma-separated values file and consisted of three columns filled with generated data containing stock purchase transactions: Name, Quantity, Price (one file line matching one table row). These entries are not unique, but single transactions that a person, stored in the Name column, may have conducted, extended by the Quantity of the stocks purchased and the Price per stock. To accommodate these structures, I created three classes with the following pseudo-description:

1. RowStore:
   (a) One row is contained in a **struct Row [Name, Quantity, Price]** object
   (b) The table consists of a **vector<Row>** object containing such rows
2. ColumnStore:
   (a) Each column in contained in a **vector<>** object depending on the column type
   (b) The table consists of a **struct Table [vector(Name), vector(Quantity), vector(Price)]**
3. HybridStore:
   (a) One partial row is contained in a **struct MiniRow [Quantity, Price]** object

(b) The table consists of a **vector(Name)** object for the first column, and a **vector<MiniRow>** for the second “column” containing (Quantity, Price) per entry

These objects would enable C++ to allocate the data internally the same way as in the definition of each approach from the previous chapter, allowing the execution of further operations over them by accessing them similarly to the various database stores. Thus, to complete the implementation of this approach, the operating functions should also be included. As such, for each of these classes I implemented functions simulating: inserting a row, inserting multiple rows, selecting rows, and three aggregation operations: **SUM(Quantity)**, **AVG(Price)**, and **SUM(Quantity * Price)**.

For initially preparing the input data before loading them into these objects, I first implemented a general function to read the CSV file and store them in a generic **vector<vector<string>>** object without specific datatypes, similar to a staging table in a database.

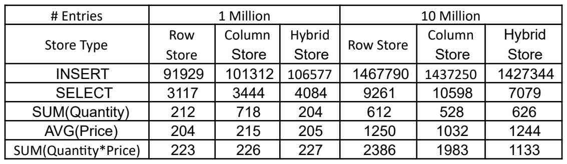
## 3.2 Evaluation

The implementation described above would theoretically access the data in three different ways to perform the required insertions and aggregations. Measuring the times elapsed for each of these operations would result in a table of these values, allowing us to compare the differences and notice the eventual advantages.

All the results depicted on this paper were conducted on a Lenovo ThinkPad machine with a Intel-Core i7-4500U processor with two cores at a frequency of 1.80 GHz and a single 8 GB DDR3 RAM, running a 64-bit version of Microsoft Windows 10 Pro.

However, once the CSV file is loaded, C++ stores these objects in-memory to be used during runtime. Consequently, since Random Access Memory (RAM) is considerably faster than massive storage alternatives, this would cause the execution of this approach to require a long time for the initial loading and allocation of the data into the objects, but a much shorter time to execute the aggregations. Testing the operations with an initial dataset of 1000 rows was almost unmeasurable in milliseconds.

Increasing the dataset size to 1 million, 10 million, and 20 million would simply result in a longer inserting time but still very quick aggregations. In order to achieve a balance between these phases, I tried simulating a slower system through the use of a Virtual Machine (VM). In this configuration, I limited the total RAM available at 3 GB and the number of used CPU cores to a single one, which I limited further down to ca. 30% of its capacity. Although the CPU specifications differ depending on the model, this serves simply to show the decreased capacity without going into further technical detail. Executing the project on this VM resulted in a longer loading time but also in a considerably longer aggregation time as expected.



| # Entries | 1 Million | | | 10 Million | | |
|---|---|---|---|---|---|---|
| Store Type | Row Store | Column Store | Hybrid Store | Row Store | Column Store | Hybrid Store |
| INSERT | 91929 | 101312 | 106577 | 1467790 | 1437250 | 1427344 |
| SELECT | 3117 | 3444 | 4084 | 9261 | 10598 | 7079 |
| SUM(Quantity) | 212 | 718 | 204 | 612 | 528 | 626 |
| AVG(Price) | 204 | 215 | 205 | 1250 | 1032 | 1244 |
| SUM(Quantity*Price) | 223 | 226 | 227 | 2386 | 1983 | 1133 |

Figure 6: Execution times for 1 million and 10 million entries (in milliseconds). Source: own representation.

Figure 6 shows two runs using two datasets of respectively 1 million and 10 million entries. Since new transactional data is normally delivered in rows, independently of the way it is stored on the database, the results show that an **INSERT** operation of these rows happens faster if they are stored in a row store, where the information remains together as is. A **SELECT** operation on all the available data is also faster on a row store since the table is recreated easily by sequentially reading the stored rows and outputting them one after the other.

On the other hand, although the environment was slowed-down artificially, the aggregation operations resulted in highly varying measurements even for repeated executions of the same dataset. Additionally, the comparison of these times with each other did not always provide results as expected. It is easily noticeable on the left that all three operations were faster on the row store object, which should not be the case. Although the simulated store objects should be stored sequentially in memory by C++ as expected, there may be multiple other environment factors causing these outcomes, including the internal behavior of the physical memory, operating system, and C++ among others when accessing the data. Even the **INSERT** and **SELECT** operations may sporadically return unexpected values, especially in a faster system because of such external factors.

This approach still provides a good logical simulation of the different storage structures used in databases because it creates an analogy for the underlying memory allocation of the data through the use of objects matching this sequential structure. Regarding the aggregation operations however an alternative approach was necessary, as will be described in the next section.

## 3.3 Approach

Since the environment factors cannot be completely avoided in such a simple simulation project, I tried to minimize their influence, especially the access method by which the data is retrieved in order to perform the aggregations. To achieve this, I implemented a completely different approach which gets already stored data and deals only with reading and aggregating operations. The input data in this case consist of one CSV file for each store mode: row, column, and hybrid (Figure 7).

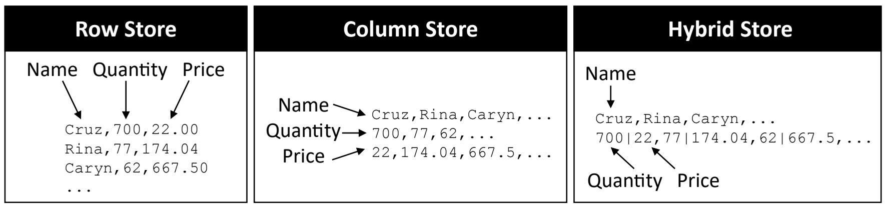


Figure 7: : Input CSV file structure for the second approach. Source: own representation

The data in each of these files is already prepared to be stored as would be expected for each store in sequence. As such, the C++ objects are not used in this case to simulate their structures. Different separators are used in these files to help the functions identify each value: "**,**", "|", but also the special new line character "\**n**", which is from the C++ stream reader perspective just another character to recognize. Thus, although the CSV files are visually shown as having multiple rows on a text editor, they are in fact a single sequence of continuous characters, just as they would be stored in memory. These CSV files serve as the "storage" unit of the database, and provide two advantages:

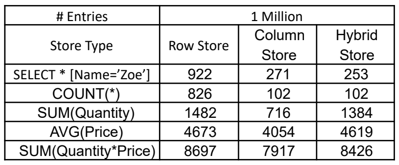
1. Accessing the data is slowed down when performing aggregations because this way the content is not loaded in-memory but continuously from a slower storage.

2. The environment factors and further internal optimizations when dealing with known objects as in the previous case of C++ structures are minimized because the structure of the data that will be read is unknown, and thus no internal optimization can be made in-between.

Since the input data is already delivered in the required form for each store method, there is no **INSERT** operation included in this approach. On the other hand, I implemented the following operations: **SELECT *** (with a filter on Name), **COUNT(*)**, **SUM(Quantity)**, **AVG(Price)**, **SUM(Quantity*Price)**. Each of these operations is computed from a separate function for each store type. These functions operate on a lower level technical approach, because the raw data coming from the CSV files needs to be parsed manually and in a different way for each respective store type. The previously described separators are identified in order to separate the values and extract the necessary information for performing the aggregation. This way the reading phase of the operation approximates a real-life database implementation better.

## 3.4 Results

The first thing to notice from the results of the second approach, some of which are shown in Figure 8 for a batch of 1 million entries, is that the aggregation operations required a longer time to compute this time, without the need of a slowed-down VM. As such, it was easier to compare the differences between their execution times even by using the smaller dataset.



| # Entries | 1 Million | | |
|---|---|---|---|
| Store Type | Row Store | Column Store | Hybrid Store |
| SELECT * [Name='Zoe'] | 922 | 271 | 253 |
| COUNT(*) | 826 | 102 | 102 |
| SUM(Quantity) | 1482 | 716 | 1384 |
| AVG(Price) | 4673 | 4054 | 4619 |
| SUM(Quantity*Price) | 8697 | 7917 | 8426 |

Figure 8: Second approach execution times for 1 million entries (in milliseconds). Source: own representation.

The **COUNT(*)** operation was considerably faster for the column and hybrid stores, since they directly count the amount of sequential values for a column, whereas in row store the reader needs to go through the whole row to count the next one. Depending on which column should be counted, the hybrid store may also need longer in other cases, but for my example implementation I use the first column to count the entries, which could be considered similar to a **NOT NULL** column in a database.

The **SUM(Quantity)** and **AVG(Price)** operations were faster on the column store because the values necessary to perform these operations are located in single respective columns and thus directly and sequentially accessible. On the other hand, the hybrid store operations took longer because they have to traverse at least one more value beside the required one, namely Price when computing SUM(Quantity) and Quantity when computing AVG(Price). These aggregations needed even more time to execute for row store because the Name column also needed to be traversed, making again the whole row.

The **SUM(Quantity*Price)** operation was a bit faster on the column store than the hybrid store, different than expected because the Quantity and Price values are stored near each other and are located sequentially one after the other directly by the reader. This aggregation should actually take longer on the row and column stores because of the additional effort to traverse unnecessary values or locate both values respectively. These measurements may come as a result of the internal C++ behavior combined with the parsing logic, whose process, although theoretically trying to approximate a real implementation, may not be fully reproduced and e.g. the value separating may affect performance.

Similarly, the **SELECT** operation also took longer on the row store than the column and hybrid store. The main reason for these results may also be supposed to be as before because of the parsing logic. However, this also provides a good opportunity to create an analogy to indexing in databases. The column and hybrid stores save the index of the filtered entries based on the first column, and then extract the values of the other columns using this index. The row store would normally also be faster if it were indexed in a real database.

## 4 Discussion

In the two sample implementations described earlier I tried to achieve an approximation of the storage and access alternatives for databases based on the gathered information from the literature research. By developing a low-level process for inserting and processing data through simple operations, I was able to simulate the behavior of these approaches and experience this process first-hand and collect some rough statistics to accompany the literature findings. Implementing a proper and extensively functional solution was out of the scope of this project, and would require considerable efforts to develop and test all the features identified for each of the store types. Compression methods, dictionaries, and complex operations or large relations and tables were thus not experimented upon.

The comparison of execution times collected from both implemented approaches did not always return the expected results. There were multiple factors which cause these differences, starting with the environment behavior. Depending on the current status of the system and the internal processes and optimizations that may occur during runtime, the performance may be affected. Furthermore, the implemented logic used in both approaches may differ slightly or highly depending on the complexity of the operation it is trying to simulate. However, it is important to note that actual databases may also differ on various scenarios depending on the implementation approach and optimization strategies followed by each vendor, as well as the context of usage.

Lastly, tuning is another factor that needs to be taken into consideration, not only when initially designing and deploying a database, but for each project or object that will make use of such instances. The data modeler and architect would need to know the features provided by the system in order to be able to optimize it for serving the requirements of the company. All these factors were highly simplified for the purpose of my sample implementation, and some of the reasons were already mentioned when discussing the results, however it enabled the analysis of the basic advantages and disadvantages of each store type.

## 5 Conclusion

In this paper I presented the importance of MMDB in today's data processing world. It is obvious that different needs led to a development of different storing layouts, each benefiting from different advantages and suffering from different disadvantages.

Row store is an intuitive layout and easy when it comes to implementation. This layout leads to faster insertion, updates and selection of tuples, but would slow down when accessing or operating on individual attributes of each tuple on the table. In my implementation I showed that this is not always the case, as different factors mentioned in the section above may play a factor.

Column store is, compared to the traditional row store, a relatively new layout. The situations in which column store outperforms the more traditional row store by storing attributes in different tables rather than whole tuples are analytical queries, OLAP processes, data compression etc. In my implementation I showed that the theoretical expectations mostly match the practical implementation.

In practical use it is not simple to choose between row store and column store, since each has its own advantages and disadvantages and are, in most cases, almost mutually exclusive. This led to the development of hybrid store, which stores the data relying mostly on inserts and updates in row store and the data mostly used for analytical queries and reads on column stores.

In the last part of the paper I implemented a simple C++ visualization of each storage layout discussed earlier and showed how expected results from the research matched or otherwise differed from my practical implementation. Finally I provided a discussion explaining the relation between my example and an actual database provider by analyzing the various factors causing such results.

## References

[1] Alfons Kemper and André Eickler. *Datenbanksysteme. Eine Einführung.* 10. Auflage. De Gruyter. pp. 583-600. 2015.

[2] Daniel J. Abadi. *Column-Stores vs. Row-Stores: How Different Are They Really?* pp. 967-980. 2008.

[3] Daniel J. Abadi. *Column-oriented Database Systems.* pp. 1664-1665. 2009.

[4] Hong Min and Hubertus Franke. *Improving In-Memory Column-Store Database Predicate Evaluation Performance on Multi-Core Systems.* pp. 63-70. 2010.

[5] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. *A Hybrid Row-Column OLTP Database Architecture for Operational Reporting.* pp. 1-14. 2009.

[6] Daniel Bernau, Olga Mordvinova, Jan Karstens and Susan Hickl. *Investigating influence of data storage organization on structured code search performance.* pp. 247-250. 2011.

[7] Anuradha S. Kanade and Arpita Gopal. *Choosing Right Database System: Row or Column-Store.* pp. 16-20. 2013

[8] Philipp Rösch, Lars Dannecker, Gregor Hackenbroich and Franz Färber. *A Storage Advisor for Hybrid-Store Databases.* pp. 1748-1758. 2012.

[9] Hasso Plattner. *A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database.* pp. 1-7. 2009.

[10] Amir El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang and George A. Mihaila *Column-Oriented Query Processing for Row Stores.* pp. 67-74. 2011.

[11] Hector Garcia-Molina and Kenneth Salem. *Main Memory Database Systems: An Overview.* pp. 509-516. 1992

[12] Mohammed Al-Kateb, Paul Sinclair, Grace Au and Carrie Ballinger. *Hybrid Row-Column Partitioning in Teradata ®.* pp. 1353-1364. 2016