Extend column-oriented execution model to Databass

Yonghe Zhao, Ziming Ma, Zhihao Zhang
{yz3687,zm2337,zz2692}@columbia.edu
Columbia University

ABSTRACT

This paper aims to explore the querying performance difference between row-oriented and column-oriented stores by extending Databass, which has a row-oriented query executor. The main work is to extend the Databass execution model to operate over columnar chunks. Some column-oriented optimization strategies, including late materialization and column compression, will be applied to further improve the performance. Several experiments based on a subset of TPC-H will be performed to both row-oriented and column-oriented Databass. The results show that the extended column-oriented Databass can improve the performance of some query executions on TPC-H by 10 to 1000 times. Furthermore, both late materialization and column compression can improve the query performance of the column-oriented model under certain conditions.

ACM Reference Format:
Yonghe Zhao, Ziming Ma, Zhihao Zhang . 2020. Extend column-oriented execution model to Databass. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The column-oriented DBMS is now the most popular architecture in OLAP scenarios e.g. data warehouse. Unlike the traditional row-oriented DBMS, it stores and processes tables in the unit of the column. Analytical queries always care about only a few of the table attributes. Thus, the column-oriented database engine only needs to fetch and operate on those useful attributes to get the correct results. The disk I/O, memory, and CPU overhead are then reduced.

In this semester's homework, we know the Databass, an experimental database written in Python with a volcano-style row-oriented execution model. For this project, we try to extend Databass to support columnar querying execution to understand better the modern column-oriented DBMS more than just reading papers.

The extension has two aspects, one for the memory storage layout, one for the query executor. Changing the memory storage layout from row/tuple-oriented tables to columns is not tricky. A straightforward way is to store the columns separately and additionally mark every attribute record with its original row index. In this way, the query executor can treat the columns independently and logically stitch them to a table correctly by the index. However, only using the columnar storage without further modifying the execution model may degrade the query performance, compared to the traditional row-oriented database [3]. The reason is that, although the number of attributes to load reduces, the overhead to reconstruct the tuple arises. This performance degrade problem becomes obvious as the projectivity increases [2].

Therefore, apart from introducing a columnar storage layer, we should further extend the execution model of Databass. By borrowing the one-column-at-a-time principle [5], each operator's input and output should be a column-based intermediate table rather than tuples. Then, every operator could exploit the benefits of columnar execution. Each operator in a query plan only load and process the columns it needs. The processing speed of queries that involve a few columns, which are the common OLAP queries, can be significantly enhanced. For example, filtering with an equal condition on a column can be efficiently done without accessing all attributes.

[1] points out the performance enhancement from column-based DB highly relies on optimizations. Several techniques are utilized to optimize its performance. In this paper, we focus on two of them: late materialization and column compression.

The original column execution implementation, based on early materialization, does not make full use of the structural advantages of the column store. Thus, we consider the late materialization. The connection between columns in a table depends on indexes, which naturally hint position information in LM. However, implementing late materialization correctly and efficiently is challenging. In this project, we will research it.

Based on the feature of the items in a column, a bunch of ways can be used for compression to simplify the computation and better utilize the locality. In this project, due to limitations from experiment's fairness and simplification of implementation, the dictionary encoding is the compression method when there are few distinct values in a column. However, another column compression may lead to better performance, like bitmap-based encoding. Thus, we will research them in the future.

2 RELATED WORK

In recent years, the column-oriented system is a hot-spot in the database domain. The column-storage architecture could effectively improve read query execution by column-at-a-time, join optimization, compression, and late materialization. There are quantities of papers focusing on each part of optimizations. Due to read-only query processing in Databass, besides columnar's design itself, we will mainly discuss read optimizations and omit the sacrifice for the update operation.

2.1 Columnar Store

C-Store [7] proposed a shared-nothing read-optimized database based on column compression, but meanwhile, it maintains a WS-RS design for unfrequent write. Appended or updated data periodically

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

transfer from WS to RS via tuple movers. After that, C-Store recompresses updated and new segments and then stores them on disk. Moreover, C-Store describes "projection" and "join index" concepts for further read queries' improvement and recovery from up to k-replica failure. Peter Boncz[5] introduces MonetDB to fit modern hardware. Its key difference from C-Store is one-column-at-a-time. Additionally, it iterates the traditional volcano-style DBMS (tuple-at-a-time) to batch processing and exploits the advantage of late materialization.

2.2 Late Materialization

Late materialization is applied in others' work [2–4] to prove the performance of column stores. Experiments in the paper by Abadi et al. [3] shows that the late materialization can reduce the query runtime by over 50%, which more remarkable than the other column-oriented optimizations. In another paper, Abadi et al. [4] mentions more details about applying late materialization to C-Store. Our work on using late materialization to prove Databass' performance will be based on some ideas from the above work.

2.3 Column Compression

Column-based compression can dramatically reduce space cost, and operating directly on compressed data benefits the query execution speed. C-Store mentions three kinds of compression algorithms, including run-length encoding, bitmap encoding, Frame of Reference encoding. After surveying several papers, D. Abadi [1] summarizes the three approaches and additionally mentions the patching technique to handle the case that the value range becomes huger or has outliers. [8] mentions several methods could improve the performance of bitmapped execution, which gives us a direction to enhance the performance of column-oriented Databass further in the future. [8] compared multiple bitmap compression algorithms with several inserted list compression and analyzes the pros and cons of the two strategies.

3 APPROACH OVERVIEW

The extension has two aspects, one for the memory storage layout, one for the query executor. Only changing the physical layout to be column-oriented is easy. The difficulty is how to integrate the new layout with the tree-like execution model and guarantee the performance. A correct way is to construct the row-based table in the scan operator and then pass the tuple one by one to the upper operator. This way adapts well to the original model. However, this implementation has poor performance [3] and is not extensible. Thus, we consider not to stitch the columns in the scan operators. Instead, pass the columns separately to the upper operator, which is the one-column-at-a-time strategy used by MonetDB [5]. Further optimizations can also be built on top of this strategy easier.

Late materialization is another column-oriented optimization that we apply to Databass to improve performance. The main idea is borrowed from the paper by Abadi et al. [4] . To use late materialization in Databass, we need to generate new query plans. The paper mentions two types of query plans, LM-pipelined and LM-parallel, that can be used for late materialization. We use the parallel one for Databass as C-Store does. The performance of late materialization can be further improved by using both parallel and pipelined plans, using different materialization strategies according to situations, etc. However, this is not in the scope of this paper.

Compression is a crucial feature of the column-oriented database. But several compression algorithms exist simultaneously, and DBMS decides to pick one specific encoding for each column according to its statistics and value frequency. Also, the compression approach on a column may vary with the change of its statistics. For the sake of simplifying the operators on top of compressed data, we will take the suggestion from [1] to encapsulate each data bucket. In our case, dictionary encoding is our focus and can explicitly tell location information for each value. Therefore, late materialization can exploit it to do further optimization on predicates or selection.

4 TECHNICAL DETAILS

4.1 Column-oriented Storage and Executor

In the original design, Databass loads the tables, stored in *csv* files on disk, into the memory data structure *pandas.DataFrame* [1], then converts it to a list of rows and registers it in a dictionary. The scan operator will then fetch the corresponding table from the dictionary and iterate over this table (list of rows), passing rows (within a *ListTuple*) to the upper operator.

In the new column-oriented memory layout, we chose to load the tables into the memory column by column, whose data type is *pyarrow.ChunkedArray* [2]. We modified the code of class *Database* in *db.py*, storing column chunks inside the new class *InMemoryColumnarTable*. Besides, to make full use of the columnar layout's structural advantages, we added a mechanism that lets scan operators know which attributes are needed in the current query so that they can only fetch those useful columns.

To combine this layout to the execution model, as we discussed above, the naive way is to construct tuples in the scan operators and pass them to the upper operators. This way is straightforward to be implemented.

After that, to make every operator compute directly on column chunks, we re-implemented all operator and expression classes, using the compute functions [3]. The useful column chunks are passed entirely between operators. This is a one-column-at-a-time strategy but not tuple-at-a-time pipelining. MonetDB [5, 6] is a famous column-oriented DBMS that applies this strategy. Although intermediate results may be generated at each operator, increasing the overhead, further optimizations can be easier implemented on top of this approach's design.

Figure 1 shows how a simple query, *SELECT A, B FROM TB1 WHERE C=1*, is executed under the column-oriented execution model.

4.2 Late Materialization

In our column-based Databass, a new table will be updated after each filter operation. We apply the idea of late materialization to improve the performance of our column-based executor further. The main idea of late materialization is postponing the construction of the table. Instead of updating the table after each filter operation, we construct the table after finishing all the filter operations in

[1] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
[2] https://arrow.apache.org/docs/python/generated/pyarrow.ChunkedArray.html
[3] https://arrow.apache.org/docs/python/api/compute.html

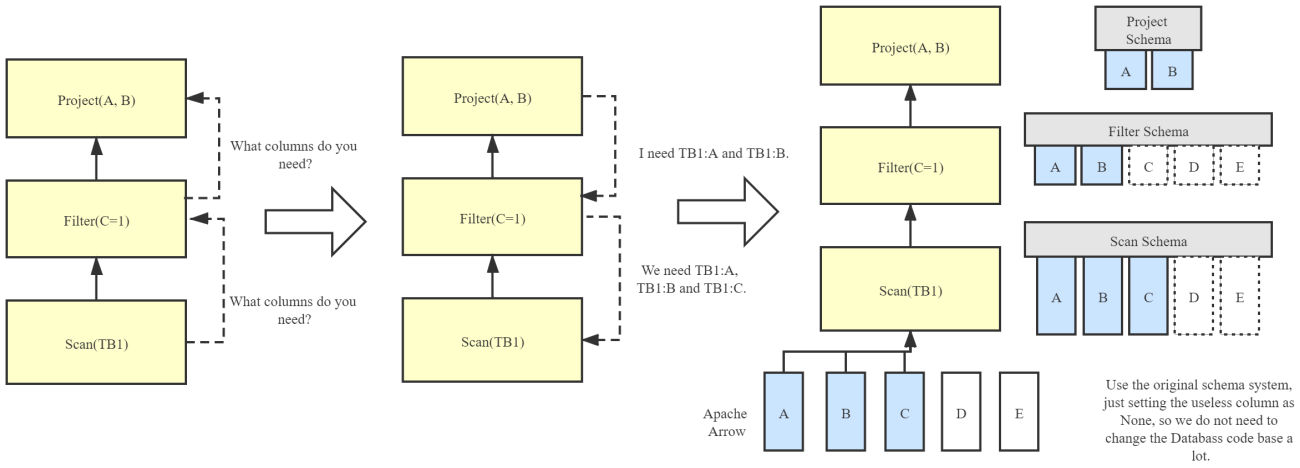

Figure 1: The procedure of executing *SELECT A, B FROM TB1 WHERE C=1* under the column-oriented execution model

a query. As C-Store, we apply the LM-parallel query plan to our column store. The LM-parallel plan consists of three main phases:

- Get masks: Do all the filter operations in the query on corresponding columns. For each filter operation, a mask will be generated. The mask is represented by a bitmap where one indicates that the value at the corresponding position is selected and 0 indicates the value is not selected.
- Merge masks: Merge all the masks by doing AND operations. Only one mask will be left after this phase.
- Construct the table: Apply the mask to all the columns projected by the query and construct the new table.

Assume we have a table TB1, and we have a query *SELECT A, B FROM TB1 WHERE C=1 AND D=2*. The LM-parallel plan for this query is shown in Figure 2. It begins with two parallel filter operations, one for C=1 and the other is for D=2. Both filter operations will scan certain columns, apply the predicate, and generate a mask represented by a bitmap. Then the AND operator will do an intersection between two masks and generate a new mask. The new mask will be applied to column A and column B to construct a table as the result of the query.

Although the benefit of the plan optimizer could be diminished by applying the LM-parallel query plan, we still expect that using LM could further improve our column-oriented executor's performance under some conditions. Late materialization can eliminate the cost of some unnecessary table construction caused by select operators. Besides that, late materialization can amplify the benefits of compressed data. Instead of decompressing and constructing tuples at an early stage, late materialization allows the executor to operate on compressed data and defer table construction.

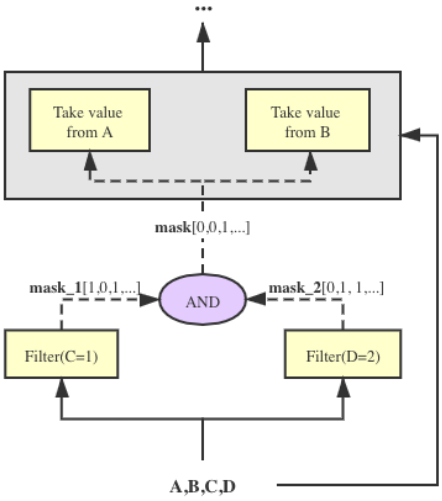

Figure 2: LM-parallel Plan

4.3 Column Compression

We will mainly discuss the implementation of column compression in Databass and how to operate directly on encoded data.

First, rather than talking about compression algorithms' details, we may want to know the high abstract object for columnar storage in the memory layer. Like mentioned before, these objects contain compression information and pass these to operators. By this means, operators could process hands-on batch data in a targeted way. Inside Apache Arrow's table class, columnar storage comprises tons of buckets, each of which stores fixed-length values defined during table loading. The initial intuition is to make each column storage unit object aware of compression-related information, such as encoding type, storage length after encoding. By this means, each operator can adjust its behavior according to different encoding types.

After figuring out the generic operation on top of compressed data, we will dive into the details of several encoding strategies. Generally speaking, there are five different kinds of compression, including run-length encoding, bitmap encoding, Frame of Reference encoding, dictionary, and the patching technique via extending

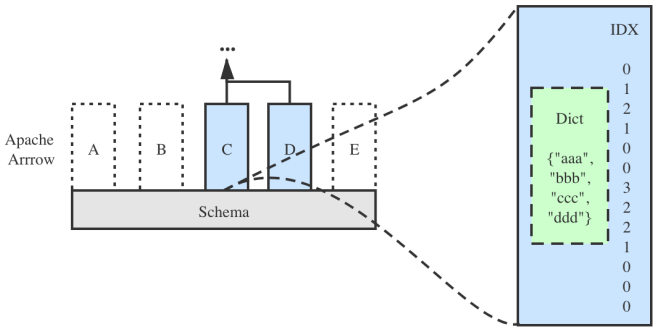

Figure 3: Dictionary Encoding Storage Structure

the combination of dictionary and FOR methods. For simplifying the reproduction of implementation, we only plan to pick a subset of these encoding approaches. After analysis of our LM-version system, we figure out the performance improvement also comes from not only column execution but also PyArrow's computation functions that directly work on Arrow's datatype. For the sake of maintaining the advantage and excluding the side effect from removing PyArrow built-in computation on comparison compression, the dictionary encoding becomes the only option. There is a naive dictionary encoding version in PyArrow, and the extension for the naive implementation is needed.

Figure 3 demonstrates the storage structure for dictionary encoding. Simply assuming column C column D are both encoded, the storage of column C consists of a dictionary and an index column in physic memory. When unique values are few and each value consumes comparably more space than an index, dictionary encoding can speed up the execution and reduce the storage cost. In our implementation, we leverage statistics information about columns to trade off whether a column should be dictionary-encoded or not, such as unique value rate and column data type.

Following the previous table in Figure 3, we use a dummy query, *SELECT COUNT(*) FROM ORDER WHERE C = "aaa", D > "2020-09-08"*, to illustrate the column-oriented execution on encoded columns.

When predicates realize they are operating on encoded columns, each predicate only compares the dictionary according to its condition expression and then creates a dictionary mask that masks which indices of the dictionary are valid. Therefore, an extra conversion can quickly be conducted for acquiring the corresponding column mask. Then the same LM procedure generates an output table.

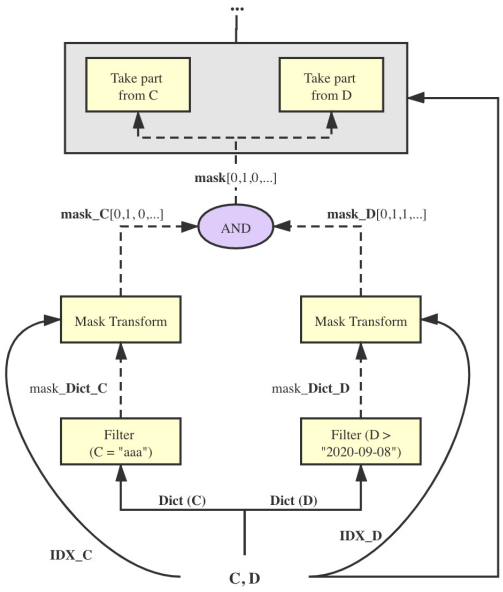

Figure 4: Encoding Column-based Execution Process

5 EXPERIMENTS

We expect to introduce the column-oriented execution model with late materialization and column compression correctly into Databass. We expect to show by test that the column-oriented execution model performs better than its row-oriented predecessor in querying speed. Furthermore, both late materialization and column compression can improve the query performance of the column-oriented model.

5.1 Setup

We setup our benchmark system on Microsoft Azure virtual machine with two 2.60GHz Intel(R) Xeon(R) Platinum 8171M CPUs and 8GB RAM. All parts of columnar Databass and old-version Databass run on the top of Ubuntu Ubuntu 18.04.5 LTS. The old Databass and columnar Databass are both implemented in Python. The Python version on benchmark system is 3.6.9.

After consulting previous works, we believe the TPC-H dataset is the right choice as the gold standard. Meanwhile, considering the trade-off between running time and tests' effectiveness, We will take CStore's[7] 7 queries on TPC-H-small for performance comparison. The queries will use three tables from the TPC-H dataset: CUSTOMER (1,500 rows, 8 columns), LINEITEM (60,175 rows, 16 columns) and ORDERS (15,000 rows, 9 columns).

5.2 Performance Comparison

We ran the following seven queries [4], similar to those on the C-Store paper, on different stages of Databass and then measured the time needed to finish the queries.

[4]For Q4, Q5 and Q6, *max(L_SHIPDATE)* is changed to *count(L_SHIPDATE)* since the Apache Arrow library does not provide a valid API to get max/min in a timestamp *ChunkedArray*.

Q1:
SELECT L_SHIPDATE, count(1)
FROM LINEITEM
WHERE L_SHIPDATE > date('1995-10-15')
GROUP BY L_SHIPDATE
Q2:
SELECT L_SUPPKEY, count(1)
FROM LINEITEM
WHERE L_SHIPDATE = date('1995-10-15')
GROUP BY L_SUPPKEY
Q3:
SELECT L_SUPPKEY, count(1)
FROM LINEITEM
WHERE L_SHIPDATE > date('1995-10-15')
GROUP BY L_SUPPKEY
Q4:
SELECT O_ORDERDATE, count(L_SHIPDATE)
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE > date('1995-10-15')
GROUP BY O_ORDERDATE
Q5:
SELECT L_SUPPKEY, count(L_SHIPDATE)
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE = date('1995-10-15')
GROUP BY L_SUPPKEY
Q6:
SELECT L_SUPPKEY, count(L_SHIPDATE)
FROM LINEITEM, ORDERS
WHERE L_ORDERKEY = O_ORDERKEY AND O_ORDERDATE > date('1995-10-15')
GROUP BY L_SUPPKEY
Q7:
SELECT C_NATIONKEY, sum(L_EXTENDEDPRICE)
FROM LINEITEM, ORDERS, CUSTOMER
WHERE L_ORDERKEY = O_ORDERKEY AND O_CUSTKEY = C_CUSTKEY AND L_RETURNFLAG = 'R'
GROUP BY C_NATIONKEY

The results are shown in Table 1. These queries are run 10 times to make the results more trusted. All measurements are in seconds. As we can see, the naive implementation to combine the column-oriented layout with the row-oriented executor degrades the performance for a little bit, the same as the conclusion in [3]. The column-oriented execution model can accelerate the performance of running these seven queries by 10 to 1000 times. However, in this experiment, the late materialization optimization does not influence the performance too much.

5.3 Experiment for Late Materialization

As we mentioned at the end of Section 4.2, using the LM-parallel query plan could diminish the plan optimizer's benefit. This is why late materialization does not obviously improve the performance of the column-oriented Databass. To further explore the performance benefit that could be brought by late materialization, we did the following experiments.

	Row	Naive EM	Column EM	Column LM
Q1	5.1724	5.8596	0.1489	**0.1473**
Q2	3.2207	3.7118	0.0034	**0.0033**
Q3	5.4665	6.0545	0.0709	**0.0692**
Q4	9.6813	10.7712	**0.2578**	0.2583
Q5	6.7756	7.6607	**0.0808**	0.0820
Q6	9.7263	10.6960	**0.1508**	0.1528
Q7	6.4581	7.3272	0.1536	**0.1516**

Table 1: Performance of row-oriented, column-oriented with early materialization and column-oriented with late materialization Databass on seven queries.

Our first hypothesis is thus: increasing the selectivity of a query will increase the cost of the column-based Databass at a larger rate than the cost of the column-based Databass with late materialization. We generated a table which contains 100,000 rows and 40 columns and ran a workload of queries on the generated table. Each query has the same number of predicates but different selectivity. The followings are some queries in the workload:

Q1:
SELECT col1, col3, col29
FROM generated_tb
WHERE col29 < 1900.0 AND col29 < 1800.0 AND col29 < 1700.0 AND col29 < 1600.0 AND col29 < 1500.0 AND col29 < 1400.0 AND col29 < 1300.0 AND col29 < 1200.0 AND col29 < 1100.0 AND col29 < 1000.0
Q2:
SELECT col1, col3, col29
FROM generated_tb
WHERE col29 < 2900.0 AND col29 < 2800.0 AND col29 < 2700.0 AND col29 < 2600.0 AND col29 < 2500.0 AND col29 < 2400.0 AND col29 < 2300.0 AND col29 < 2200.0 AND col29 < 2100.0 AND col29 < 2000.0
...
Q10:
SELECT col1, col3, col29
FROM generated_tb
WHERE col29 < 10900.0 AND col29 < 10800.0 AND col29 < 10700.0 AND col29 < 10600.0 AND col29 < 10500.0 AND col29 < 10400.0 AND col29 < 10300.0 AND col29 < 10200.0 AND col29 < 10100.0 AND col29 < 10000.0

Figure 5 shows the result of running the above queries on the column-based Databass and the column-based Databass with late materialization. The x-axis is the selectivity of each query, and the y-axis is the running time (in second) of executing the queries. The result shows that the column-based Databass with late materialization is stable to the increasing of selectivity, and it has a better performance than the column-based Databass without late materialization when selectivity is high.

Besides of the selectivity, the number of columns to project in a query could also influence the performance of late materialization.

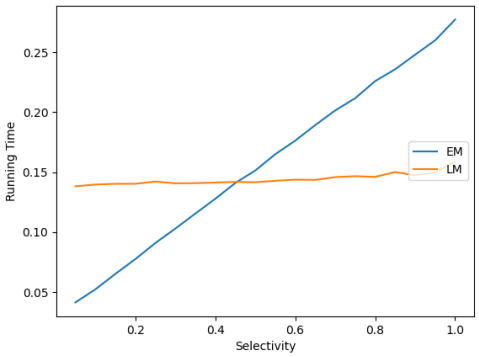

Figure 5: Performance comparison between EM and LM based column-oriented executor at different selectivity

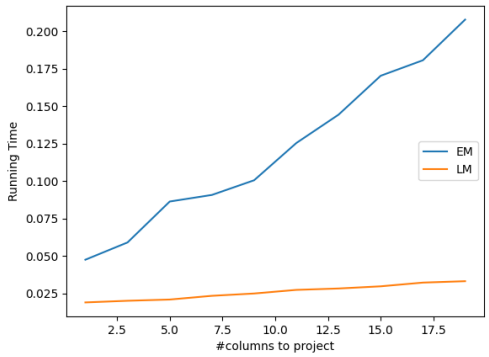

Figure 6: Performance comparison between EM and LM based column-oriented executor at different number of columns to project

We produce another experiment to prove our second hypothesis: increasing the number of columns to project of a query will increase the cost off the column-based Databass at a larger rate than the cost of the column-based Databass with late materialization. We run another workload of queries on the generated table, each query has the same selectivity (0.9) and the same number of predicates (10) but different number of columns to project. The followings are some queries in the workload:

Q1:
SELECT col0, col1
FROM generated_tb
WHERE col29 < 9900.0 AND col29 < 9800.0 AND col29 < 9700.0 AND col29 < 9600.0 AND col29 < 9500.0 AND col29 < 9400.0 AND col29 < 9300.0 AND col29 < 9200.0 AND col29 < 9100.0 AND col29 < 9000.0

Q2:
SELECT col0, col1, col2, col3
FROM generated_tb
WHERE col29 < 9900.0 AND col29 < 9800.0 AND col29 < 9700.0 AND col29 < 9600.0 AND col29 < 9500.0 AND col29 < 9400.0 AND col29 < 9300.0 AND col29 < 9200.0 AND col29 < 9100.0 AND col29 < 9000.0

...

Q10:
SELECT col0, col1, col2, col3, col4, col5, col6, col7, col8, col9, col10, col11, col12, col13, col14, col15, col16, col17, col18, col19
FROM generated_tb
WHERE col29 < 9900.0 AND col29 < 9800.0 AND col29 < 9700.0 AND col29 < 9600.0 AND col29 < 9500.0 AND col29 < 9400.0 AND col29 < 9300.0 AND col29 < 9200.0 AND col29 < 9100.0 AND col29 < 9000.0

Figure 6 shows the result of running the second workload of queries. The x-axis represents the number of columns to project in the query, and the y-axis is still the running time of the query. The result confirms the hypothesis. With the increase of the number of columns to project, the running time of the column-based Databass increases at an obviously higher rate than the running time of the column-based Databass with late materialization.

With the above two experiments, we can conclude that, in general, using late materialization can improve the performance of our column-based Databass, especially when the selectivity and number of columns to project a query are high.

5.4 Experiment for Compression

Following the description in Section 4.3, compression in our implementation will only be applied when two conditions are simultaneously met. The one is that the data type is one of string and date string, and the other is unique value rate ($\frac{\#unique\_value}{\#num\_row}$) is less than a threshold. As we expect, the dictionary encoding could speed up the query execution when filtering by a compressed column. With this hypothesis, we leverage the same dummy table in experiment 5.3. Specifically speaking, the type of the last 10 columns are all strings, and each of them consists of the same 40 unique strings. We design a straightforward workload for the LM and encoding-based LM version to conduct the comparison test. The following workload adjusts the number of predicates:

Q1:
SELECT col1, col3, col5, col7, col9, col11, col30
FROM generated_tb
WHERE col30 = st1_1

Q2:
SELECT col1, col3, col5, col7, col9, col11, col30
FROM generated_tb
WHERE col30 = str_1 AND col31 = str_2

...

Q10:
SELECT col1, col3, col5, col7, col9, col11, col30
FROM generated_tb
WHERE col30 = str_1 AND col31 = str_2 AND col32 = str_3 AND col33 = str_4 AND col34 = str_5 AND col35 = str_6 AND col36 = str_7 AND col37 = str_8 AND col38 = str_9 AND col39 = str_10

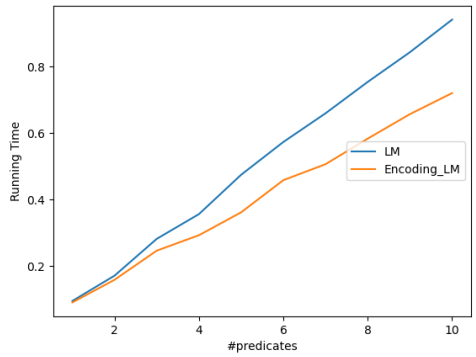

Figure 7: Performance comparison between LM and encoding-based LM column-oriented executor at different number of predicates

Figure 7 shows the result of running the above queries on the LM Databass and the LM Databass with encoding. The x-axis is the number of predicates for each query, and the y-axis is the running time (in second) of executing the queries. The result shows that further encoding optimization could achieve a better performance than the raw LM Databass. Moreover, with the predicates' amount rising, the gap of execution time becomes larger. Thus, our hypothesis is proven valid.

6 CONCLUSIONS

In this paper, we describe two ways to extend the column-oriented query execution model to the row-oriented Databass. We find that using the one-column-at-a-time strategy like the MonetDB is better in performance and extensibility. On top of that, we explain how to apply the late materialization and column compression to optimize the query executor further. Finally, we design a set of experiments to show that the column-oriented execution model performs 10 to 1000 times better than its row-oriented predecessor in querying speed. Furthermore, both late materialization and column compression can improve the column-oriented model's query performance under certain conditions.

REFERENCES

[1] D. Abadi, P. Boncz, S. H. Amiato, S. Idreos, and S. Madden. *The design and implementation of modern column-oriented database systems.* Now Hanover, Mass., 2013.

[2] D. J. Abadi, P. Boncz, and S. Harizopoulos. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2012.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? *SIGMOD*, 08:967–980, 2008.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. *2007 IEEE 23rd International Conference on Data Engineering*, 2007.

[5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. 5:225–237, 2005.

[6] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35, 01 2012.

[7] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented dbms. *VLDB*, 05:553–564, 2005.

[8] M. Velez, J. Sawin, A. Ingerson, and D. Chiu. Improving bitmap execution performance using column-based metadata. *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2016.