Support Roaring Bitmap in DuckDB

Nuo Xu
xunuo@usc.edu
USC Viterbi School of Engineering
Los Angeles, USA

Qihan Zhang
qihanzha@usc.edu
USC Viterbi School of Engineering
Los Angeles, USA

Wanen Gao
wanengao@usc.edu
USC Viterbi School of Engineering
Los Angeles, USA

ABSTRACT

In this project, we aim to support Roaring Bitmap as a new type of index into the DuckDB internal, version 0.10.1 [3]. It's integrated both in the filter (WHERE clause) as well as Hash Join to accelerate query processing. Our result shows that enabling Roaring Bitmap in DuckDB gives out 2% - 16% performance gain on various workloads, and hence it's quite feasible for DuckDB to optimize its design and performance using Roaring Bitmap.

KEYWORDS

Roaring Bitmap, Indexing, DuckDB

1 INTRODUCTION

As data volumes continue to grow exponentially, the demand for high-performance Online Analytical Processing (OLAP) database systems has become more pronounced. Among such systems, DuckDB has emerged as a prominent open-source solution, noted for its ability to efficiently handle large analytical queries directly on compressed data. Despite its robust performance in many scenarios, DuckDB currently supports a limited range of indexing mechanisms—specifically, the Min-Max Index (Zonemap) and the Adaptive Radix Tree (ART) Index [4]. These indexes, while effective under certain conditions, do not sufficiently address the challenges posed by scenarios that would benefit from the high query efficiency of bitmap indexes, especially in cases involving low cardinality data.

Bitmap indexes, which represent data sets using bit sequences where each bit denotes the presence or absence of an item, are particularly adept at accelerating analytical queries on large datasets. They excel in environments where queries frequently involve multiple conditions that need to be evaluated across vast arrays of data. For example, in querying a database for all graduate students who have taken a specific course, bitmap indexes can rapidly perform logical operations on bitmaps representing different query conditions, thus significantly speeding up data retrieval.

However, the traditional bitmap index can be inefficient in terms of space, especially when dealing with sparse data sets, as it tends to consume significant storage for sequences composed predominantly of zeroes. To address this limitation, the Roaring Bitmap has been proposed as an advanced form of bitmap index. Roaring Bitmaps optimize storage and performance by using variable-sized containers to manage data density, thus maintaining high compression ratios and low memory usage while still ensuring rapid query execution. These capabilities suggest that the Roaring Bitmap could be an invaluable addition to DuckDB, enhancing its suitability for a broader range of OLAP tasks.

This paper proposes the integration of Roaring Bitmap indexing into DuckDB to expand its indexing capabilities. The introduction of this new index type aims to leverage the inherent strengths of Roaring Bitmaps—namely, their efficiency and scalability—to enhance DuckDB's performance in handling complex analytical queries over large datasets. By doing so, DuckDB can better meet the demands of modern data analytics, providing faster query responses and more efficient data processing.

Our approach involves a detailed study of existing index implementations in DuckDB, followed by the design and implementation of the Roaring Bitmap index. We will evaluate the integration of this new index through rigorous benchmarks using standard OLAP workloads, comparing the performance of the modified DuckDB against its original configuration. This research will contribute valuable insights into the feasibility and benefits of extending DuckDB with Roaring Bitmap technique, potentially setting a precedent for future enhancements in similar database systems.

2 INDEXING IN DUCKDB

Currently, DuckDB has two types of indexes: Zonemaps and ART indexes. More details are shown as below.

2.1 Zonemaps

DuckDB automatically creates zonemaps (also known as min-max indexes) for the columns of all general-purpose data types [4]. These indexes are used for predicate pushdown into scan operators and computing aggregations. This means that if a filter criterion (like WHERE column1 = 123) is in use, DuckDB can skip any row group whose min-max range does not contain that filter value (e.g., a block with a min-max range of 1000 to 2000 will be omitted when comparing for *= 123* or *< 400*).

2.1.1 The Effect of Ordering on Zonemaps. The more ordered the data within a column, the more useful the zonemap indexes will be. For example, in the worst case, a column could contain a random number in each row. It's unlikely for DuckDB to skip any row groups. The best case of ordered data commonly arises with DATETIME columns. If specific columns will be queried with selective filters, it is best to pre-order data by those columns when inserting them. Even an imperfect ordering will still be helpful.

2.1.2 Microbenchmark: The Effect of Ordering. For an example, let's repeat the microbenchmark for timestamps with a timestamp column that is sorted using an ascending order vs. an unordered one.

Table 1: The Effect of Ordering

Column type	Ordered	Storage size	Query time
DATETIME	yes	1.3 GB	0.6 s
DATETIME	no	3.3 GB	0.9 s

The results (Table 1) show that simply keeping the column order allows for improved compression, yielding a 2.5x smaller storage size. It also allows the computation to be 1.5x faster.

2.1.3 Ordered Integers. Another practical way to exploit ordering is to use the INTEGER type with automatic increments rather than UUID for columns that will be queried using selective filters. UUIDs will likely be inserted in a random order, so many row groups in the table will need to be scanned to find a specific UUID value, while an ordered INTEGER column will allow all row groups to be skipped except the one that contains the value.

2.2 ART Indexes

DuckDB allows defining Adaptive Radix Tree (ART) indexes in two ways [4]. First, such an index is created implicitly for columns with PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints. Second, explicitly running the CREATE INDEX statement creates an ART index on the target column(s).

The trade-offs of having an ART index on a column are as follows:

- It enables efficient constraint checking upon changes (inserts, updates, and deletes) for non-bulky changes.
- Having an ART index makes changes to the affected column(s) slower compared to non-indexed performance. That is because of index maintenance for these operations.

Regarding query performance, an ART index has the following effects:

- It speeds up point queries and other highly selective queries using the indexed column(s), where the filtering condition returns approx. 0.1% of all rows or fewer. When in doubt, use EXPLAIN to verify that your query plan uses the index scan.
- An ART index has no effect on the performance of join, aggregation, and sorting queries.

Indexes are serialized to disk and deserialized lazily, i.e., when the database is reopened, operations using the index will only load the required parts of the index. Therefore, having an index will not cause any slowdowns when opening an existing database.

3 BITMAP INDEXING IN DBMS

Bitmap Indexing is a data indexing technique used in database management systems (DBMS) to improve the performance of read-only queries that involve large datasets. It involves creating a bitmap index, which is a data structure that represents the presence or absence of data values in a table or column.

In a bitmap index, each distinct value in a column is assigned a bit vector that represents the presence or absence of that value in each row of the table. The bit vector contains one bit for each row in the table, where a set bit indicates the presence of the corresponding value in the row, and a cleared bit indicates the absence of the value.

3.1 Bitmap Index Structure

A bitmap is the combination of two words: bit and map. A bit can be termed as the smallest unit of data in a computer and a map can be termed as a way of organizing things.

Bit: A bit is a basic unit of information used in computing that can have only one of two values either 0 or 1. The two values of a binary digit can also be interpreted as logical values true/false or Yes/No.

Bitmap Indexing is a special type of database indexing that uses bitmaps. This technique is used for huge databases when the column is of low cardinality and these columns are most frequently used in the query. It uses a compact binary representation to store the occurrence of each value or combination of values in each attribute, allowing for fast, set-based operations.

3.2 Features of Bitmap Indexing in DBMS

Space efficiency Bitmap indexes are highly space-efficient because they use a compact binary representation to store the occurrence of each value or combination of values in each attribute. This makes them especially useful for large datasets with many attributes.
Fast query processing Bitmap indexes can be used to quickly answer complex queries involving multiple attributes using set-based operations such as AND, OR, and NOT. This allows for fast query processing and reduces the need for full table scans.
Low maintenance overhead Bitmap indexes require relatively low maintenance overhead because they can be updated incrementally as data changes. This makes them especially useful for applications where the data is frequently updated.
Flexibility Bitmap indexes can be used for both numerical and categorical data types, and can also be used to index text data using techniques such as term frequency-inverse document frequency (TF-IDF).
Reduced I/O overhead Bitmap indexes can be used to avoid expensive I/O operations by using a compressed representation of the data. This reduces the amount of data that needs to be read from the disk, improving query performance.
Ideal Choice Bitmap indexing is a powerful technique for efficiently querying large datasets with many attributes. Compact representation and set-based operations make it an ideal choice for data warehousing and other applications where fast query processing is critical.

3.3 Roaring Bitmap

The roaring bitmap data structure [1] is commonly used for analytics, search, and big data projects due to its high performance and compression ratio. Compressing big integer sets while maintaining quick access to individual elements is a significant strength of roaring bitmap. It internally uses different types of containers to accomplish this.

A roaring bitmap is a set of unsigned integers consisting of containers of disjoint subsets. Each subset has a 16−bit key index and can hold values from a range of size 2^{16}. This allows unsigned 32-bit integers to be stored as Java shorts, as the worst-case scenario only necessitates 16 bits to represent a single 32-bit value. The selection of the container size also ensures that, in the worst-case scenario, the container still fits in the L1 cache of a modern CPU. Figure 1 shows how a roaring bitmap structure looks.

Our integer's 16 most significant bits are the bucket or chunk keys. Each chunk of data represents the base of the range of values in the interval $0 <= n < 2^{16}$. Furthermore, the chunk won't be created if no data in the value range. In the first chunk, we store the first 10 multiples of 2. Moreover, in the second chunk, we have

	Container with the first 10 multiples of 2	Container with 100 consecutive integers starting from 2^{16} (65536)	Container with even numbers between 131072 and 196607
16 most significant bits	0000 0000 0000 0000	0000 0000 0000 0001	0000 0000 0000 0010
Cardinality(No. of integers)	10	100	2^{15} (32768)
Integer in container	0	65536	131072
	2	65537	131074
	4	65538	131076
	...	...	...
	18	65635	196606

Figure 1: Roaring Bitmap Structure

100 consecutive integers starting from 65536. The last chunk in the image has even numbers between 131072 and 19660.

4 IMPLEMENTATION

In this project, we use a C++ implementation of Roaring Bitmap [2] to optimize DuckDB's scan filter and Hash Join. For building a bitmap index on a certain column, we define a new Bitmap index using SQL command:

```
CREATE BITMAP index_name on some_table(some_column)
```

The *BITMAP* keyword will serve the same position as *INDEX*, but will create a different index class: Roaring Bitmap. Then it will go through a similar process as creating an ART index, and fill the corresponding data to the empty Roaring Bitmap owned by ColumnData objects defined in DuckDB internal. After filling in the data, each time we need to scan the data on this column(not only a single scan but can be a joint scan), it will go through the Roaring Bitmap to pre-filter the data. Multiple columns' bitmaps can be used together(like intersecting them upon an AND operator) to speed up the pre-filtering.

As for the implementation of Bitmap in Hash Join, we will add all data on the left side to an empty Bitmap associated with physical_hashjoin object, and use the data on the right side to probe this bitmap first before doing the real hash probes. since we will create a full-fledged bitmap on the fly, this will introduce some overhead. In the experiment, we find that this overhead will even shadow the benefit of prefiltering. In the future, we will still explicitly create a Roaring Bitmap before we do the Hash Join operations. This modification mainly wants to say that it's possible to use Roaring Bitmap for Hash Join Operations.

4.1 Filtering

4.1.1 How a sequential scan is performed in DuckDB. DuckDB divides data from a table into multiple RowGroups similar to Parquet [5], each holding 122880 rows of data. A RowGroup stores information about the ID of the first row it contains, total tuple count, Zonemap statistics for filtering, etc,. All RowGroups of a table are stitched together by a segment tree to perform data access with the benefit of only loading the data needed.

Internally, DuckDB uses multiple levels of ScanState to manage scans. When a scan is performed, DuckDB will initiate a scan for each RowGroup by traversing the previously mentioned segment tree. Within each RowGroup, data is processed one partition (or vector) at a time. For each partition, it will first check its Zonemap to determine whether to skip, then fetch the data to process and return. When filters are involved during this process due to the filter push-down hinted by the optimizer, target columns of those filters are scanned first, build a selection vector omits all the rows that don't satisfy the filters, and finally using this selection vector to scan and process all the rest columns.

4.1.2 Integration of Roaring Bitmap. The implementation of Roaring Bitmap integration with filtering happens mostly during the scan of a RowGroup, as shown in Figure 2. Each column is associated with a map of Roaring Bitmaps, each representing the existence of a unique value within this column. When multiple filters are used during the scan, instead of fetching and reading in the target columns of filters one by one, we simply use the bitmaps of those columns to perform bit-wise operations (intersection for AND filter, union for OR filter). The result bitmap will be used to generate the selection vector, which is then applied to all columns of the table to get the final result of the scan.

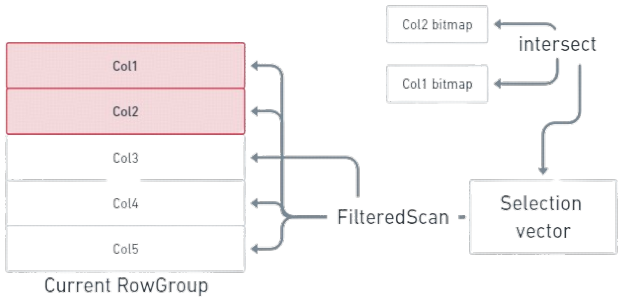


Figure 2: Control flow of table scan with bitmap

As we can see, the usage of Roaring Bitmap saves great computation overheads while generating the selection vector. This is mainly because bitmap operations like intersection and union are very cheap, and we use SIMD instructions (AVX-512 specifically) internally to further accelerate this process. Moreover, potential I/O overhead could also be avoided, especially when there are many filters applied during this table scan. This is due to the fact that we now generate the selection vector before actually fetching the data, and data partitions are more likely to be skipped if we know the selection vector in advance.

4.2 Hash Join

Different database internals will choose different strategies to select their join operators. As for DuckDB, it will always use Hash Join to do the equality join. In fact, this strategy is a bit rigid and sometimes cannot generate a good performance. After we force it to use other join operators like Nested Loop Join and Merge Join, the internal will frequently encounter exceptions for it cannot handle equality join using other join operators. So we only try to deal with the situation of Hash Join.

When the logical plan has become a physical one and is ready to do the Hash Join, in the sink phase, we use a mutex(avoid multiple threads contention) to help add the data on the left side to an empty

Roaring Bitmap associated with a HashJoinGlobalSinkState object. And then the data will be used to form a Hash Table as well. After all the pre-processes are ready, this thread will enter a pipeline mode and do the internal join. Before we do the hashing probing, we will pre-filter the data first: once there is a match found on both sides, we will break the pre-filtering and do the normal hash operations. After that, the thread will signal a "Have More Output" label to the pipeline and end itself. If there isn't any match found here, the thread will signal a "Need More Input" label to the pipeline and wait for the extra data.

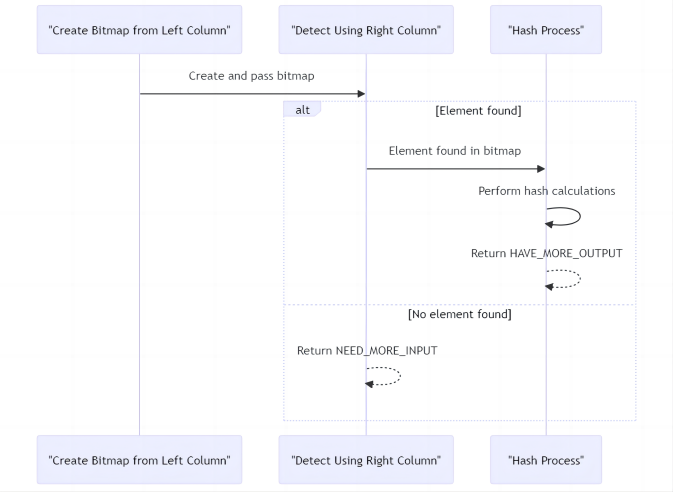


Figure 3: Sequential Graph of Bitmap for Hash Join

Figure 3 shows the process of this pipeline mode. When we actually do the physical Hash Join, it's a very late stage and it's quite hard to find a data path to the Column_Data object. But in the future, we will find a way to reuse the bitmap associated with the Column_Data to avoid the overhead of creating a Bitmap on the fly.

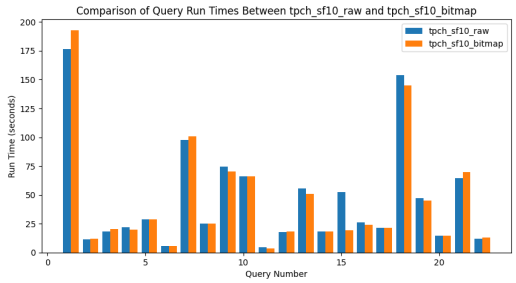
5 EXPERIMENTS

5.1 Workload and Machine settings

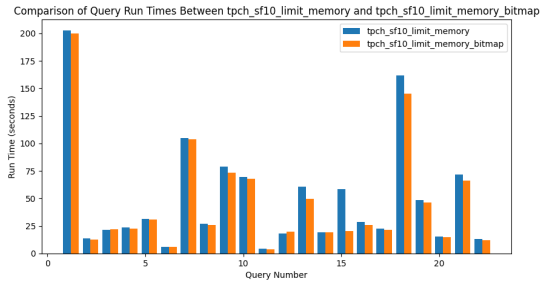
All results are collected on a Docker Container, with a 16-core 2.30GHz 11th Gen Intel(R) Core(TM) i711800H CPU and 32 GB RAM. This will give DuckDB a memory limit of 12.3 GB by default. To evaluate the performance under an Out-of-memory workload, we further reduced DuckDB memory limit to 5 GB, roughly half of the entire data size. In general, we use three types of workload in our experiments: TPCH standard 22 queries with scaling factor 10, IMDB Join Order Benchmark(JOB), and a synthesized micro-benchmark called TPCH micro focusing on multiple equality filters.

5.2 Filtering

The result from TPC-H SF10 is shown in Figure 4. We see that for pure in-memory workload, integrating bitmap indexing improves overall runtime by 3%. As the data goes out of memory, the improvement gets more significant, with an increase from 3% to 8% gain as shown in Figure 2. Especially, the performance could get



(a) In-memory workload



(b) Out-of-memory workload (5GB memory limit)

Figure 4: TPC-H Result

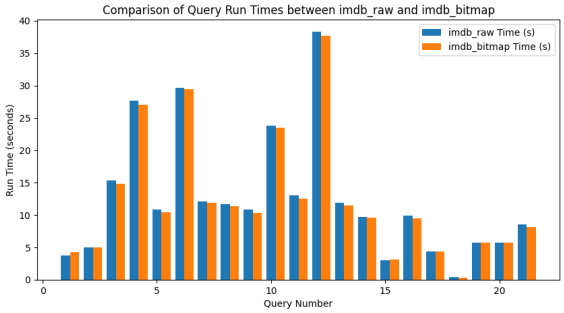


Figure 5: IMDB result

improved by at most 62% in some queries like query 15. Indeed, there are some queries that perform worse when roaring bitmap is enabled. Previously, it only took 176 seconds for query 1 to execute, while 192 seconds when roaring bitmap is introduced. This is due to the fact that TPCH query 01 only contains one inequality filter $l\_shipdate <= CAST('1998-09-02' AS\ date)$ along with multiple *Group By* and *Order By*.

IMDB JOB benchmark further strengthens the analysis. The result in Figure 5 clearly shows a trend similar to the TPCH result, with most queries having improvements from 1.44% to 25%. Such improvement may not be significant enough, however, we argue that it would become unreasonable if such small changes in a commercial database system lead to a drastic improvement.

Figure 6 shows the result of our synthesized micro-benchmark. This workload consists of 9 synthesized queries on top of TPCH

Table 2: Performance Improvement on Different Workloads

Workload	Original Time(s)	Time With Roaring Bitmap(s)	Improvement
TPC-H SF=10	1016.41	987.11	3%
TPC-H SF=10 with limited memory	1103.37	1010.56	8%
TPC-H micro	31.60	26.57	16%
IMDB JOB	5907.74	5787.39	2%

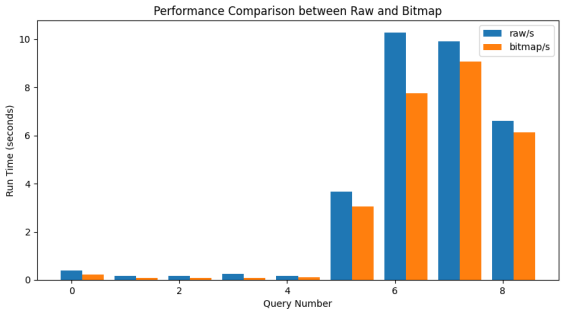


Figure 6: Micro-benchmark result

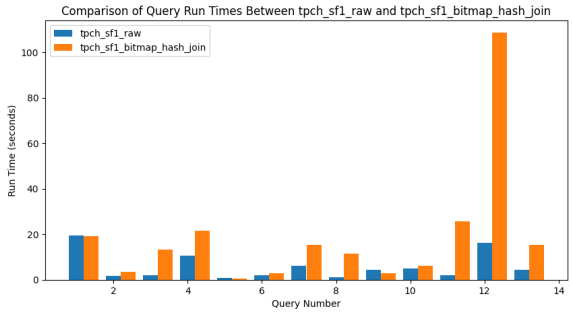


Figure 7: Hash Join Result on TPC-H SF=1

dataset. These synthesized queries mainly target heavy equality-check filters. An example of such a query would be as follows:

```
SELECT count(*) FROM lineitem, orders
WHERE
lineitem.l_linestatus = orders.O_ORDERSTATUS
AND lineitem.l_orderkey = orders.o_orderkey;
```

Note here it contains two equality filters on columns *l_linestatus* and *l_orderkey*. On average, enabling bitmap indexing gives about 16% performance gain under heavy equality filters. This corresponds with our assumption that using bitmap operations like intersecting and union-ing to generate a selection vector instead of computing it within a loop is beneficial.

The overall improvement of our work on filtering in three workloads is shown in Table 2. Certainly, Roaring Bitmap can bring some improvements especially if the memory is limited, for its ability to save a lot of overhead on frequent I/Os.

5.3 Hash Join

Figure 7 shows the result of using Roaring Bitmap on TPC-H standard 22 queries(scale factor = 1). Unfortunately, the result shows that using bitmap will introduce extra overhead. For some queries, like queries 4, 5, 6, 7, 8, 20, 21, and 22, they consume disastrous time to finish the queries, so we just skip them in the figure 7.

The reason why when we use bitmap in Hash Join will use more time is that it will create a Roaring Bitmap(filling, to be more specific) on the fly. So the creation of a bitmap will be included in the on-time queries, which will shadow the benefit of using a bitmap.

However, if we create the bitmap before the on-time queries, the result will be another story. Since the operation of Hashing Probing is still a costly overhead.

6 FUTURE WORK

In the future, we would be delighted to furnish our work and get it merged into the main branch of DuckDB. Currently, we only support varchar and integer-type data for bitmap indexing. However, it could be extended to more data types easily like the ART index. Plus, the logic of aiding Hash Join could be improved, we will try to reuse the bitmap created to avoid the costly pay of creating a bitmap on the fly. The best and most feasible way to achieve this is to first finish our work in persisting bitmap indexing on disk through serialization.

Furthermore, the selection logic of join could also be improved like PostgreSQL [6]. Instead of rigidly choosing Hash Join all the time, we could let the optimizer dynamically choose the join operators based on the physical machine settings and cardinality of each side of the join. Finally, it would be ideal to pass the information about bitmap indexing to the optimizer so that it would help build a better execution plan, e.g. pushdown more filters on columns with bitmap indexing to gain performance.

REFERENCES

[1] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Software: Practice and Experience*, 46(5):709–719, April 2015.
[2] CRoaring. https://github.com/roaringbitmap/croaring.
[3] DuckDB. https://github.com/duckdb/duckdb/tree/v0.10.1.
[4] DuckDB SQL Indexes. https://duckdb.org/docs/sql/indexes.
[5] Apache Parquet. https://parquet.apache.org/.
[6] PostgreSQL. https://www.postgresql.org/.