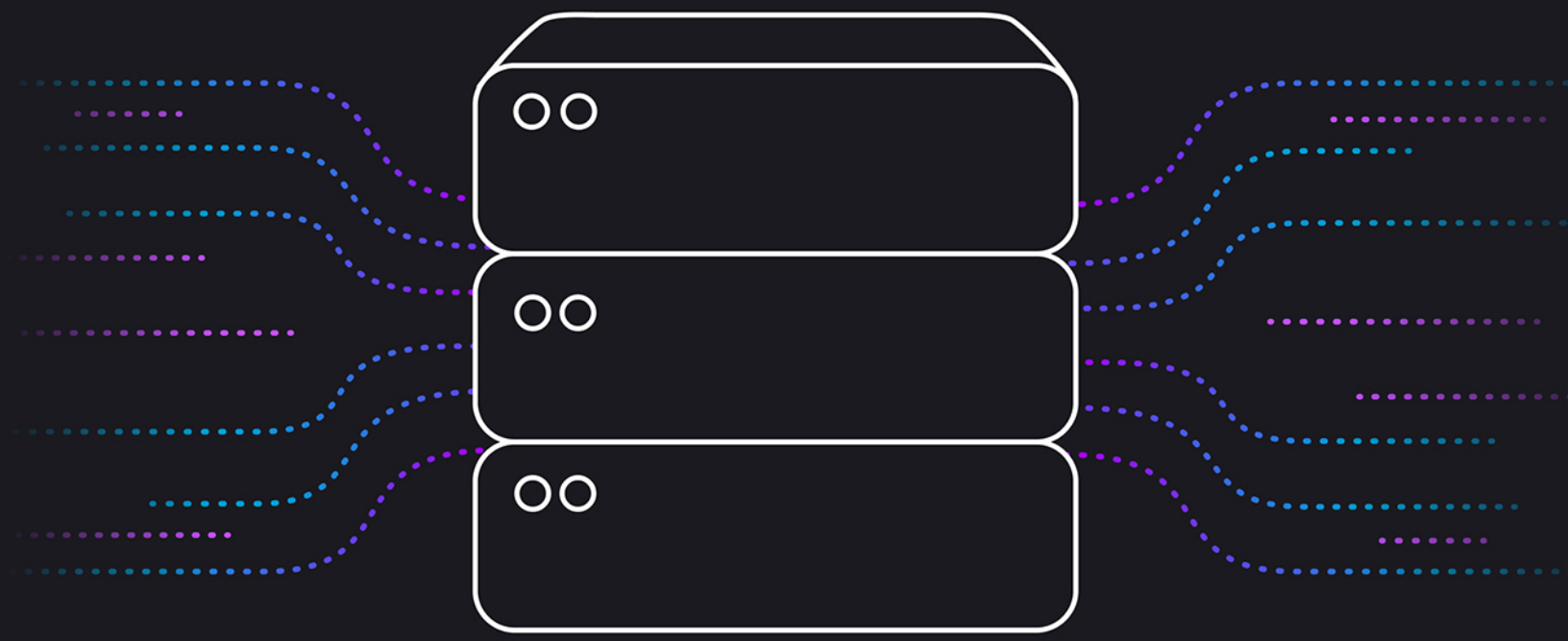
RESEARCH PAPER

BIPie: Fast Selection and Aggregation on Encoded Data Using Operator Specialization



WRITTEN BY
Michal Nowakiewicz, Eric Boutin, Eric Hanson,
Robert Walzer, Akash Katipally

SingleStore

BIPie: Fast Selection and Aggregation on Encoded Data using Operator Specialization

Michal Nowakiewicz
MemSQL

Eric Boutin
MemSQL

Eric Hanson
MemSQL

Robert Walzer
MemSQL

Akash Katipally*

ABSTRACT

Advances in modern hardware, such as increases in the size of main memory available on computers, have made it possible to analyze data at a much higher rate than before. In this paper, we demonstrate that there is tremendous room for improvement in the processing of analytical queries on modern commodity hardware. We introduce BIPie, an engine for query processing implementing highly efficient decoding, selection, and aggregation for analytical queries executing on a columnar storage engine in MemSQL. We demonstrate that these operations are interdependent, and must be fused and considered together to achieve very high performance. We propose and compare multiple strategies for decoding, selection and aggregation (with GROUP BY), all of which are designed to take advantage of modern CPU architectures, including SIMD. We implemented these approaches in MemSQL, a high performance hybrid transaction and analytical processing database designed for commodity hardware. We thoroughly evaluate the performance of the approach across a range of parameters, and demonstrate a two to four times speedup over previously published TPC-H Query 1 performance.

ACM Reference Format:
Michal Nowakiewicz, Eric Boutin, Eric Hanson, Robert Walzer, and Akash Katipally. 2018. BIPie: Fast Selection and Aggregation on Encoded Data using Operator Specialization. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3183713.3190658

1 INTRODUCTION

Hardware technology advancements have made it possible to process data at a much faster rate. This has enabled companies to make decisions on *real-time data* by sending analytical queries to a database. These queries often have an ad-hoc nature, complex filters, and tend to benefit little from pre-built indices. As such, those analytical queries tend to scan a very large volume of data. In order to save on disk and memory bandwidth, the data is often encoded to use a smaller number of bytes. This encoding offers tremendous opportunities for query performance improvement, as we demonstrate in our discussion of enhancements known collectively as Business Intelligence ProcessIng on Encoded Data (BIPie[1]).

We introduced BIPie in MemSQL 6 to improve the performance of queries accessing data stored in a columnar table. We specifically focus on queries filtering the input table (WHERE clause), grouping on one or more columns or expressions (GROUP BY clause), and aggregating one or more expression (i.e. SUM(x)). This is achieved by performing query processing on the encoded data directly by using vector processing with SIMD and by specializing operators to optimize performance for certain runtime parameters.

Operating on encoded data allows BIPie to avoid decoding and materializing decoded values. Encoded data also provides additional information available as part of the encoding.

BIPie implements multiple variants of the selection and aggregation operators, and chooses between them at runtime. Each operator implementation is optimized for a different set of parameters, such as the number of bits per value, the selectivity, the number of groups, and the number of aggregates to compute. For instance, special versions of aggregation are designed to optimize the cases (i) when there is a very selective filter, (ii) when a very small number of distinct groups is used, or (iii) when there is a high number of integer sums. These variants of operators are constructed by merging one of three selection strategies with one of three aggregation strategies.

Gather selection leverages the gather instruction on modern microprocessors and is optimized for lower selectivity. *Special Group selection* is optimized for higher selectivity and is fused with the aggregation strategy to perform selection and aggregation in the same step. *Compaction selection* is a safe fallback.

In-Register aggregation is optimized for values with a lower number of bits and lower number of groups. *Sort-Based aggregation* is optimized for lower selectivity and a large number of aggregates to compute. *Multi-Aggregation* is optimized for a larger number of aggregates to compute.

The paper is structured as follows: first we introduce some background on the MemSQL database engine and the real-time analytical workload that is optimized by BIPie (Section 2). Then, we cover an overview of the architecture of BIPie (Section 3) before diving into the details of the selection strategies (Section 4) and the aggregation strategies (Section 5). Then, we compare each combination of strategies across a broad range of parameters, and evaluate BIPie in the context of TPC-H Query 1. We compare the performance of Query 1 with other database engines.

*Work done while at MemSQL

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3190658

[1] Also, MemSQL engineers have a deep fondness of pies.

2 BACKGROUND

MemSQL is a distributed memory-optimized SQL database, which excels at mixed real-time analytical and transactional processing at scale. MemSQL can store data in two formats: an in-memory, row-oriented store and a disk-backed, column-oriented store. Tables can be created in either rowstore or columnstore format, and queries can involve any combination of both types of tables. MemSQL takes advantage of in-memory data storage with multiversion concurrency control and novel, memory-optimized, lock-free data structures to enable reading and writing data with a high concurrency, allowing real-time analytics over an operational database. The MemSQL columnstore uses innovative architectural designs to enable real-time streaming analytical workloads with low-latency queries over tables with ongoing writes [21]. Along with its scalable distributed architecture, these innovations enable MemSQL to achieve sub-second query latencies over large volumes of changing data. MemSQL is designed to scale on commodity hardware and does not require any specialized hardware.

MemSQL utilizes a shared-nothing architecture; nodes in the distributed system do not share memory, disk, or CPU. There are two tiers of nodes: scheduler nodes (called aggregator nodes) and execution nodes (called leaf nodes). Aggregator nodes serve as mediators between the client and the cluster, while leaf nodes provide the data storage and query processing backbone of the system. Users route queries to the aggregator nodes where they are parsed, optimized, and planned. The optimizations discussed in this paper will focus on execution strategies within a single leaf node. Further details about distributed query execution and optimization in MemSQL can be found in [10].

Query plans are compiled to machine code using LLVM [14] and are cached to expedite subsequent executions. MemSQL caches compiled query plans to provide the most efficient execution path. The compiled query plans do not pre-specify values for the parameters, allowing MemSQL to substitute values upon request, and enabling subsequent queries of the same structure to run quickly, even with different parameter values.

MemSQL strives to allow customers to run the MemSQL database on a wide range of hardware platforms, including on-premises and cloud installations. At the same time, MemSQL has a performance-first orientation. As such, we leverage hardware trends using state-of-the-art query execution approaches, such as the ones described in this paper.

2.1 Columnar Encoded Data

BIPie focuses on processing columnar-encoded data. This section introduces the column-store format used by MemSQL. More details can be found in [21]. The MemSQL columnstore index is split between a mutable region and an immutable region. The immutable region of the columnstore index is column-oriented and compressed. Rows can be marked as deleted in the immutable region, but cannot be updated. The mutable region is row-oriented, uncompressed, and updatable. The mutable region represents a small fraction of rows, recently added or modified. It is compressed into the immutable region by a background task. This paper focuses on query processing on the immutable region.

Rows in the immutable region of the columnstore are grouped into segments. Each column within a segment is compressed, stored, and accessed separately. All columns preserve the same order of records. A segment contains approximately one million records. Segment columns are encoded using one of multiple possible encodings. Among the supported encodings in MemSQL are: delta encoding, run length encoding, dictionary, and integer bit packing. The encodings are chosen during compression of rows based on two factors: size of the resulting compressed data, and usefulness of the encoding for query execution.

Run length encoding (RLE) is useful when it is common for the same value in a column to occur many times in consecutive rows. An encoded RLE stream consists of a sequence of pairs (value, count); the value is the uncompressed value, and the count specifies how many times the value is repeated in consecutive rows. Dictionary encoding has two components: a dictionary containing all distinct values, and a bit packed sequence of integers identifying elements in this dictionary. Bit packing encoding represents all the values in the sequence using the same number of bits. We use the smallest number of bits needed to represent the maximum index in the dictionary. The bits are concatenated into one vector without any gaps between values.

Each segment contains metadata for each of the columns, such as the minimum and maximum values that appear in each column of the segment. This metadata has two applications. The metadata allows for segment elimination during query processing when the query can determine that the filter expression on this column would reject all contained rows. The metadata is also used to determine if overflow is possible for numeric values when they are used in expressions and sums. If we can determine that no overflow can occur, then overflow checks can be avoided for the segment. In this paper, we make the assumption that it is always possible to determine that overflow is impossible within a segment, and we not discuss overflow checks during presentation of the algorithms.

MemSQL is a distributed database. Tables are partitioned based on a key specified by the user, and the partitioning properties of the data are leveraged by the query optimizer to avoid repartitioning of the data during join or aggregation, if the data is already partitioned correctly.

Query processing on the MemSQL columnstore, including all query processing algorithms described in this paper, follows a *batch processing* structure. A moving window of a fixed number of rows (up to 4096 rows in MemSQL) is used when scanning the columnstore table. The rows within this window form a batch of rows. We entirely process one batch before moving to the next one and we never revisit previous batches. This technique was previously employed in MonetDB/X100 [7].

2.2 Simplifications

In order to facilitate the presentation of the algorithms, we make several simplifying assumptions about the representation of input data and the type of database queries that process it.

First, we assume that there is a single group by column with no more than 256 unique values. We also assume that the group by column uses dictionary encoding in combination with bit packing. All distinct values are assigned consecutive integer identifiers starting

from 0. Then, the value of the group by column for every row gets replaced with a corresponding id, called *group id*. Finally, group ids are stored in a bit vector sequentially, with no gaps, using the smallest possible fixed number of bits. This encoding format allows the use of arrays indexed by group ids for updating per-group aggregate results with no need to reference the dictionary until the point of outputting aggregation results. We also assume that the order of rows is arbitrary, specifically that data is not sorted on the group by column. We assume that all aggregate columns are of integer data types no larger than 8 bytes, contain no null values, are encoded using bit packing, and do not use dictionary encoding.

Whenever we mention bit unpacking, we always mean outputting unpacked values in an array using the smallest power-of-two word size that all values fit in (1, 2, 4, 8). Using the smallest word is important for achieving optimal performance in some cases.

Those simplifications are not restrictions to the implementation of BIPie. They simply streamline the explanation for clarity. Expanding the techniques beyond these simplifications is a mechanical and straightforward extension of the techniques we describe.

2.3 Workload

This paper focuses on executing analytical workloads on encoded data stored in columnar format. Specifically, it focuses on pushing selection and aggregation into the columnar scan. BIPie efficiently executes queries of the following form directly on encoded columnar data:

```
SELECT
   g,
   count(*),
   sum(agg1), ..., sum(aggn)
FROM
   columnarTable
WHERE
   <filter expression>
GROUP BY
   g;
```

Identifiers "g", "agg1", ..., "aggn" reference columnstore columns directly. We do not discuss the evaluation of the filter expression. We assume in this paper that its result has already been computed and that its cost is small relative to the entire query cost. Filters and all aggregates are optional.

The techniques introduced by BIPie are applicable to any workload that consists of queries either directly in this form, or of queries that can be decomposed into queries of this form. A good examples of this workload is TPCH query 1, which we will use to evaluate BIPie in section 6.3.

3 BIPIE OVERVIEW

We introduced BIPie in MemSQL 6 to improve the performance of relatively simple queries accessing data stored in a columnstore index. Specifically, we targeted queries filtering the input table, grouping on one or more columns or expressions and aggregating one or more columns or expressions. The approach used to provide a performance improvement rests on three pillars: (i) operating on encoded data, (ii) using vector processing with SIMD and (iii) specialization of operators.

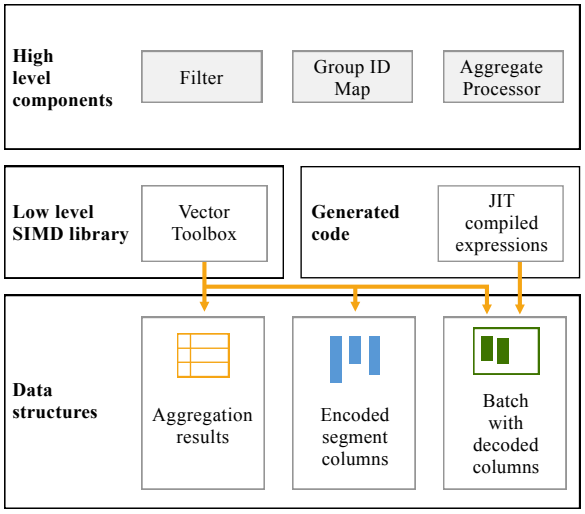


Figure 1: BIPie Architectural Overview

Operating on encoded data provides three main benefits: (i) BIPie can often avoid decoding values either fully or partially, (ii) BIPie can avoid materializing decoded values and (iii) BIPie leverages the additional information available to us as part of encoding. For example, dictionary encoding already provides the injective mapping from column values to small integers, which can be used as a perfect hashing function of that column.

Operating on encoded data without materializing decoded values provides the important benefit of processing vectors of inputs in simple loops that repeat the same predictable sequence of instructions on independent data elements present in vectors [7]. Predictability and independence inside the loop lead to a very good utilization of the CPU instructions pipeline. Data independence also means that the loops can use SIMD for parallel processing of tuples of consecutive elements.

Specialization of operators allows BIPie to outperform the previous implementation by having multiple special implementations of operators and selecting between them at runtime. Each operator implementation is optimized for different inputs and has different restrictions on the inputs it can handle. For instance, this could mean implementing special versions of the aggregation operators. One implementation is used only with a very selective filter. Another implementation is used for a very small number of distinct groups. Another implementation is used when there are a large number of integer sums to compute.

The choice of aggregation method occurs at run-time, and the method can change from segment to segment. The choice of the selection method can change from batch to batch, and is based on the actual selectivity calculated after evaluating the filter for the batch.

Figure 1 illustrates an overview of the components of the MemSQL columnstore scan. Specifically, it focuses on the single-segment scan, with the main data structures it operates on. Segment management, segment elimination, and deleted row management are omitted for clarity.

BIPie integrates decoding, filtering, and aggregation with grouping into the columnstore scan. The scan combines elements of just-in-time compilation, vectorization, and SIMD processing. The implementation is separated into three layers: (i) the Vector Toolbox, (ii) the generated code, and (iii) the high-level components containing operator logic.

The Vector Toolbox is a library of low level vector functions. The Vector Toolbox is highly optimized, has versions compiled for different generations of CPUs that can be automatically switched at run-time based on the hardware that the product is running on, and has no dependencies on any other code of the engine. Vector Toolbox functions can operate on decoded and encoded data.

The generated code is used, among other things, for all scalar expressions in the query, including expressions that are part of a filter, grouping expressions or inputs to aggregates. Generated functions always operate on decompressed column data. This is essential to maintain a low query compilation time, otherwise each expression would have to be compiled for each possible combination of encodings.

Finally, there is a higher level logic of the columnstore scan organized into classes responsible for decoding, filtering, grouping, and aggregation. The high level logic orchestrates execution, choosing at run-time between multiple implementations of the same operation, calling low-level functions from other layers.

The filter component evaluates the filter expression on a columnar-oriented batch of records, combines the result with information about deleted records, and produces a selection vector indicating which records are selected by the query.

Aggregation is decomposed into two components: the Group ID Mapper and the Aggregate Processor. The Group ID Mapper takes in the group by columns specified in the query and produces a single vector of integer group ids. The Group ID Mapper replaces the hash table lookup step in a classical implementation of aggregation. It leverages optimization opportunities present when operating on encoded data. For example, dictionary encoded data provide the group id mapper with a perfect collision-free hashing. Exploiting this property provides a significant performance improvement. The Aggregate Processor takes in a group id vector and a selection vector produced by the Filter component, and computes the aggregates for each group. The Aggregate Processor chooses among the many aggregation strategies implemented in the vector toolbox at run time.

We select an aggregation strategy for each segment. The aggregation strategy for each segment is selected based on the maximum number of groups in the segment as calculated from the segment metadata and the number and type of aggregates.

The paper focuses on the part of VectorToolbox that is used by the Aggregate Processor. We will show multiple ways of using SIMD to accomplish the task of the Aggregate Processor and we will look at how each of these methods is useful for different kinds of inputs. These methods are not universal, but they can add significant speedups in many specific, but useful and common, cases of queries.

4 SELECTION

After evaluating the filter expression for rows in the columnstore table being scanned, we get a byte vector, called the *selection byte vector*. The selection byte vector marks positions of elements that should be removed with a byte value of zero, while the remaining positions have the value 0xFF. This is consistent with how AVX2 comparison instructions store the output for single byte elements. Further, in order to exclude deleted records from being processed, we write a zero in the selection byte vector position for each deleted record in the batch.

Another form of selection vector that we sometimes use is a *selection index vector*, which is an array that contains the ordinal positions of qualifying rows. The high-level idea of selection considered in this paper is to remove unwanted rows from processing and leave the remaining data from columns in a form that can be further processed without the need to reference the selection byte vector or selection index vector.

Selection can naively be implemented using a conditional branch instruction dependent on the filter result. This approach severely limits data level parallelism, e.g. SIMD. In addition, it limits the effectiveness of the CPU pipeline, since the CPU cannot accurately predict which instructions to execute next. In BIPie, the selection operator avoids conditional branching dependent on the filter result. This makes the code path very predictable for the CPU, and makes very efficient use of CPU pipelining. The absence of diverging code paths for adjacent rows also makes it possible to efficiently leverage SIMD.

4.1 Compacting operator

The compacting operation takes two inputs: (i) a selection vector and (ii) an input vector. The compacting operator produces an output vector. The selection vector contains a byte per value of the input vector. It is very efficient to produce this representation through a vectorized execution of the filter expression.

Compaction is equivalent to the following sequential pseudocode: a counter C is initialized to zero. The operator iterates through the selection vector. Whenever the selection vector is zero, C is incremented. When the value is non-zero, we write to the output vector and increment C. Depending on the mode of operation of the compaction operator, a different value is written. In *index vector* mode, the counter C is written to the output vector. In *physical compaction* mode, the C^{th} value from the input vector is copied to the output vector. The *physical compaction* mode requires the input vector to be unpacked, and element sizes must be a power of two.

A data-parallel and branch-free version of this code can efficiently be implemented in SIMD [20]. Both variants of the compaction operation use between 0.4 and 0.6 CPU cycles per row in the CPU cache.

4.2 Gather Selection

Gather selection works in two steps. The first step is to use the *index vector* mode of the compacting operator to transform the selection vector into a selection index vector. The second step starts by iterating through the selection index vector. For each index, we fetch a word containing the bit packed value from the encoded column. We then extract the actual value from this word.

Fetching, extracting, and storing unpacked values can be done entirely using data-level parallelism with the AVX2 instruction set. Specifically, fetching can use a gather instruction that can load multiple array elements for a given sequence of indices into multiple lanes of a SIMD register. The second step of this operation needs to be repeated for every group by column and aggregate column involved in the query. It combines bit unpacking and removing filtered-out rows.

Table 1 presents the performance, in CPU cycles per row, of gather selection for different bit widths. As expected, the performance slows down as the bit width increases because fewer elements can be packed in a SIMD register.

Bit width of input column	5	10	20
CPU cycles per row	1.08	1.33	1.63

Table 1: Gather Selection Performance

The main difference between the gather selection and the compaction operation in physical compaction mode is that the gather selection operation only unpacks values that are selected, while the 'physical compaction' mode of the compacting operator requires the entire input column to be unpacked.

4.3 Selection by Special Group Assignment

It is very common to combine selection and aggregation in a single query. In fact, the query shape that BIPie targets combines selection and aggregation. In many cases, very few rows are rejected by the filter. The special Group Assignment Selection technique is designed to optimize this scenario by fusing together the filtering component and the group id mapping component. As such, in contrast to other selection methods, Selection by Special Group Assignment returns a group id map.

The following observation motivated this approach to selection. We were running a simple SQL query of the form

```
SELECT a, sum(x) FROM t WHERE b = 1 GROUP BY a
```

where both columns "a" and "b" had a small number of distinct values. We noticed that this query is visibly slower than

```
SELECT a, b, sum(x) FROM t GROUP BY a, b
```

even though the result of the first query is the subset of the rows output by the second one, and can be computed in post-processing with negligible cost. In the second query of the example, we replaced the filter expression with a new group by column that was introduced to represent all rows that should be rejected.

In the first query, the gather selection uses an indexed read to fetch the value of the column x. Fetching the column x requires reading an address calculated based on the selection vector. This limits the algorithm's ability to leverage the pipelining of the CPU. The second query executes a sequential scan of the column, has a more predictable memory access pattern, and leverages CPU pipelining more efficiently.

The idea of the proposed method is to create a special, unused group index. For each position that is not set in the selection byte vector (for each filtered out row), we assign the same unused group index. Next, the chosen aggregation method processes all the rows regardless of filter, and using the modified group ids. As the result is outputted, we discard the results for the special group that was created just for the purpose of filtering. This process can also be seen as pushing grouping and aggregation ahead of parts of the selection operation in the processing pipeline.

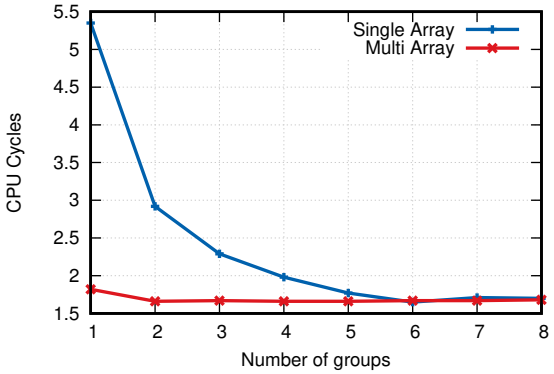


Figure 2: CPU cycles per row for COUNT aggregation

5 GROUPING AND AGGREGATION

After the selection strategy has been applied, we need to compute the aggregates. This is done by combining a *group id map*, which indicates the group id each record belongs to, with an input batch. An in-depth description of how to generate the group id map in the general case is outside of the scope of the paper. In simple cases, for example, when grouping by a single dictionary encoded column, the group id is simply the dictionary id. After the group id map and the selection vector has been produced, an aggregation strategy is applied to compute the aggregate (i.e., compute the SUM of a value for each group). To achieve truly high performance, this step has to be optimized for the underlying hardware, and adapted to the parameters of the data. The parameters of the data include the number of groups, the number of aggregates being computed, the number of bits of each value being aggregated, and even the selectivity. This section first covers the naive *Scalar Method* and evaluates it. Then, we go on to cover Sort-Based SUM Aggregation, In-Register Aggregation and Multi-Aggregate SUM Aggregation.

5.1 Scalar Method

We start with an analysis of the naive scalar (no SIMD) implementation of group by aggregation with sum. Algorithm 1 demonstrates a naive strategy to execute a single sum.

Algorithm 1 Scalar Aggregation

```
for (int i = 0; i < number_of_rows; ++i)
    sum[group_column[i]] += sum_column[i];
```

Figure 2 (Single Array) illustrates the CPU cycles per row for the scalar implementation. Interestingly, for a very small number of groups this code runs noticeably slower than for six or more groups (2.9 CPU cycles per row with two groups vs 1.65 cycles per row with six groups). This likely happens due to the high chance of

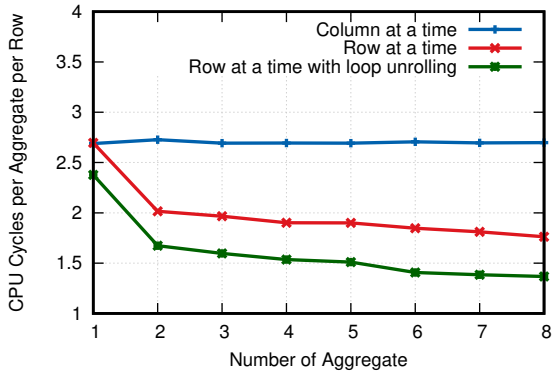


Figure 3: Comparison of Scalar Sum Implementations

adjacent rows trying to make updates to the same memory location, resulting in CPU pipeline stalls. Similar effects can be observed even with a large number of groups whenever there is a high frequency group index in the input column. This can happen with a partially sorted group by column or when there is data skew. The simple fix is to unroll the aggregation loop twice or more, and use two or more arrays with sums, switching between them in a round robin manner for consecutive rows and finally merging the partial results from all these arrays at the end of the computation.

If there are multiple sums in the same query, then aggregation can be done either one column at a time or one row at a time. In the first case we would fully process the first aggregate column for a batch of rows before moving on to the next column. In the latter case, we would update all sums for a single row before moving on to the next row. The second method, with the row-oriented layout of aggregation results in the array, performs better. Figure 3 shows results in CPU cycles per row per aggregate for 32 groups and varying number of sums using both methods. Additionally, it shows the impact of unrolling the inner loop over all columns in the row-at-a-time variant.

5.2 Sort-Based SUM Aggregation

Sort-Based Sum Aggregation is our first aggregation strategy using SIMD. This strategy requires sorting of the row indices within each batch of rows by the group index for the row. The resulting sorted array can be viewed as a concatenation of N sub-arrays for N groups, some of which are potentially empty, and each containing indices of all rows within a batch that fall into the same group. Once the batch is sorted appropriately, we compute sums for each group, one aggregate column at a time and one group at a time. We fetch the aggregate column values for a given group by iterating through the row indices, and compute the sum. We utilize the SIMD gather instruction to optimize fetching the aggregate column values given an array of row indices.

We use bucket sort for sorting, each bucket corresponding to a single group. First, we compute the number of items in each group, just as if a COUNT(*) aggregation were part of the original query. If a COUNT(*) aggregation is part of the query, we only compute it once and reuse the results. This gives us the number of rows per bucket, from which we get sub-ranges of the output array that

	1 sum	2 sums	4 sums
4 groups	3.13	2.21	1.74
8 groups	3.59	2.49	1.89
16 groups	3.61	2.48	1.92

Table 2: CPU cycles per row per aggregate for Sort-Based SUM Aggregation

correspond to different groups. In the second pass over the data, we append each row index to the sub-range associated with its group. For a small number of groups, it is possible to get write conflicts for bucket counters for adjacent rows, resulting in performance penalties similar to what we mentioned in section 5.1 in the case of scalar group by sum. We avoid write conflicts by using two separate counters for each bucket, one for even and one for odd rows. This technique provides the same effect as the the multi-array aggregation of section 5.1.

When summing values from aggregate columns, we perform decoding, or bit unpacking, on the encoded inputs as the values are aggregated. The decoding, selection, and aggregation are performed together in one optimized unit. Rows filtered out are excluded from the array of sorted row indices. If gather or compacting selection is used, the rows are excluded before sorting. In the case of selection by special group assignment, the rows are rejected during the sorting.

The Sort Based SUM Aggregation is a good fit for scenarios with a combination of low filter selectivity and high number of aggregates. The extra cost of sorting is fixed regardless of the number of aggregates, and aggregate processing after sorting does not require any extra steps in the presence of filters. Table 2 shows the measured costs of running this strategy expressed in CPU cycles per row per aggregate, with varying number of groups and sums (with 23 bit-packed aggregate columns and no filters). The table shows that the cost per aggregate reduces with the number of aggregates, as the fixed sorting cost is amortized over a larger number of aggregates. This method works with aggregate columns in their raw bit-packed, non-filtered representation. The other aggregate methods presented require all inputs to be decoded first. This underrepresents the performance of this approach, as the decoding cost is not included in results shown for other methods below.

5.3 In-Register Aggregation

In-register aggregation is the second SIMD-friendly strategy for computing aggregates that we present in this paper. It is based on the idea of keeping intermediate results entirely in CPU registers instead of in memory. The technique applies to both count and sum, but is limited to small number of groups, up to around 32 on today's hardware.

Each aggregate in this approach is processed separately. To understand the data layout, we use an example in which we are computing the row count per group. There are N groups. Group indices each encoded using 1 byte, and are all in the range 0 to $N-1$.

As the input rows are processed sequentially, the group ids are loaded into a SIMD register. The i^{th} lane (byte) of this register is holding data for the i^{th} row in the vector. Each lane of the register uses its separate virtual array of row counts per group. N SIMD registers are used to store all the virtual arrays. Each register keeps

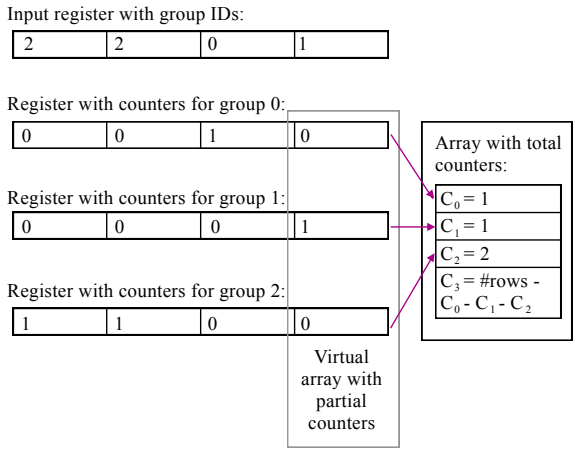


Figure 4: In-Register Aggregation Data Layout

Algorithm 2 In-Register aggregation

```
for i=0..N-1
   mask=simd_compare(V, i)
   virtualArray[i] = simd_add(virtualArray[i], mask)

simd_compare(V, i):
   mask = [0...0]
   for each lane in V:
      if V[lane] = i:
         mask[lane] = 0xFF
      else:
         mask[lane] = 0
   return mask

simd_add(v1, v2)
   result = [0...0]
   for each lane in v1:
      result[lane] = v1[lane] + v2[lane]
   return result
```

the counts for one group. In the case of the COUNT aggregate, we can optimize away processing for the group $N-1$ by subtracting the count of each group from the total count, thus saving one register. Figure 4 illustrates the layout of virtual arrays using SIMD registers. In the figure, four lanes are used, and there are four groups.

The pseudocode to update the virtual arrays for the next vector of group id values (denoted V) is provided in Algorithm 2. For clarity, we also provide scalar equivalent code for the SIMD intrinsic used. The simd_compare primitive is used to create a *mask*, which is a SIMD register containing as many lanes as V. For each lane of V, if the content of the lane is equal to i, then each bit of the corresponding lane in the mask is 1. Otherwise, each bit of the corresponding lane in mask is 0. After computing the mask, the mask is added to the virtual array for the counter using SIMD addition.

Variant	Input size	size/counter	Instructions/32 values
COUNT(*)		4 bits	1.5
SUM(x)	1 byte	16 bits	3
SUM(x)	2 bytes	32 bits	7
SUM(x)	4 bytes	32 bits	12

Table 3: Number of instructions per group for 32 values with In-Register Aggregation

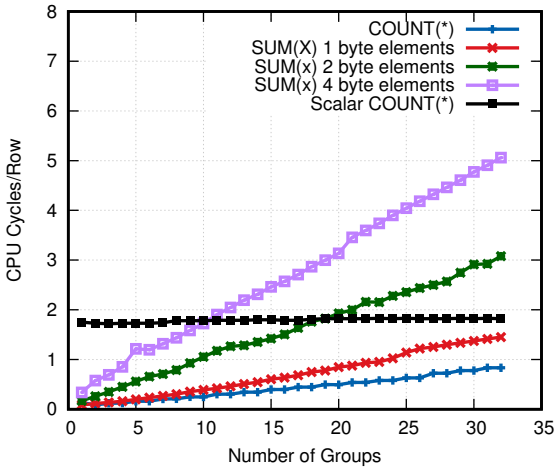


Figure 5: Performance of In-Register Aggregation

After processing a group id vector, the counters are merged into total counters. First, each lane of each virtual array has to be negated, because adding the mask (0xFF) is equivalent to adding -1, which has left the negation of the count in the lane. Then the virtual arrays are collapsed into a single counter per group. The merged results include the calculated value of the one missing counter based on the total row count.

MemSQL contains the implementation of specialized versions of this strategy for N up to 32, for count and for sum of one byte, two byte, and four byte values. The generation of these implementations is assisted by macros and the template engine present in modern compilers. The appropriate implementation is selected at runtime. It is possible to determine an upper bound on the number of groups in a batch by leveraging the information from the encoding. For example, in the case of group by of a dictionary encoded column, the number of distinct entries in the dictionary can be used as the upper bound of the number of groups.

Table 3 summarizes the number of CPU instructions per group actually used in our implementation of each variant.

Figure 5 illustrates the comparison of performance for all the variants of the grouping aggregates using virtual arrays in registers for an increasing number of groups. Scalar group by count(*) data has been added as a reference. As expected, the performance linearly degrades with the number of groups, as an operation has to be executed for each group. Queries using fewer bytes for each value perform much better, as each register has more lanes, and more SIMD parallelism can be extracted from those implementations.

Number of Sums	Sizes (bytes)	CPU cycles/row/sum
2	8-2	1.37
3	8-4-1	1.43
4	8-8-4-2	0.91
5	8-4-4-2-2	0.77
5	4-4-2-2-2	0.75

Table 4: Sample Performance of Multi-Aggregate

5.4 Multi-Aggregate SUM Aggregation

Multi-Aggregate SUM Aggregation is the third SIMD-friendly aggregation strategy presented in this paper. This strategy applies to queries with multiple sums. Unlike the previous two methods, this strategy uses data-level parallelism horizontally instead of vertically; multiple aggregates for the same input row are processed together, instead of multiple input rows for the same aggregate.

Multi-Aggregate SUM can also be seen as the evolution of the scalar group by sum implementation for more than one aggregate column from section 5.1. We have shown in section 5.1 that row-at-a-time aggregation for multiple sums is faster than column-at-a-time aggregation. It could be further improved if we packed inputs for multiple sums for the same row into one SIMD register and executed only one set of load-add-store instructions for all of them.

Our inputs are stored column-wise in memory. In order to assemble rows we need to reorganize values from the columns being summed. We consider a simple case of four 64-bit input columns aggregated using 64-bit arithmetic. After loading one 256-bit SIMD vector worth of data (four rows) from each of the four columns, we get a 4x4 matrix stored in four registers. Transposing this matrix will give us the desired layout of the data in these registers. This can be done in eight AVX2 instructions (four PUNPCKLQDQ and four PUNPCKHQDQ instructions).

In the general case, there can be different numbers of input columns and they can store elements of different byte sizes. This poses a challenge for the transposition. BIPie uses a composition of template functions to create specialized SIMD implementations that cover all interesting cases. BIPie expands 1 and 2-byte input values to 4-bytes and everything larger to 8-bytes when generating output. We do this to guarantee that we can safely sum up to 65536 rows using 64-bit additions in SIMD lanes without getting an overflow for input values of up to 4-bytes (8-byte elements must rely on other methods for overflow checks). BIPie supports an arbitrary number and combination of sizes of input columns as long as, after the expansion, all elements for a single row can fit into a 256-bit SIMD register, with 32-bit expanded elements being 32-bit aligned, and with 64-bit elements being 64-bit aligned. An example of value packing is shown in Figure 6. In the example, the letters "A", "B", "C", "D" and "E" denote column names, while the following index corresponds to the position of an entry within that column. Each row in the figure represents a SIMD register.

Table 4 contains examples of performance results for 32 groups, and different numbers of aggregate columns and combinations of sizes of their elements. Results are expressed as the number of CPU cycles per row per aggregate. The table shows that the more sums are done, the higher the efficiency per sum.

Input (after loading from 5 columns to SIMD registers):

A1				A2		A3	A4
B1		B2		B3	B4		
C1	C2	C3	C4				
D1	D2	D3	D4				
E1		E2		E3	E4		

Output (after generalized transposition of data in registers):

32-bit		32-bit		64-bit		64-bit		64-bit	
C1		D1		A1		B1		E1	
C2		D2		A2		B2		E2	
C3		D3		A3		B3		E3	
C4		D4		A4		B4		E4	

Figure 6: Example of Data Layout for Multicolumn Aggregation

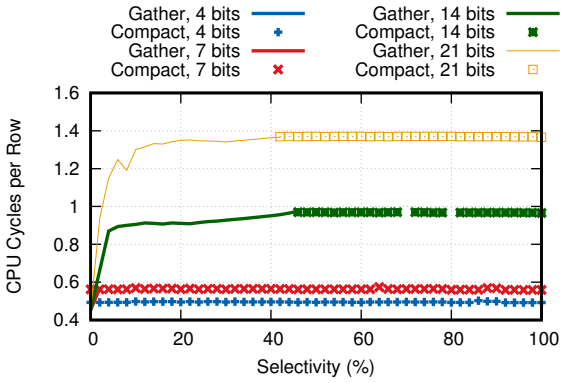


Figure 7: Comparison of Selection Strategies

6 EVALUATION

In this section, we evaluate the performance of BIPie. First, we compare selection methods. Then, we perform an in-depth comparison of all combinations of selection and aggregation methods presented, evaluating each method across a range of parameters, such as selectivity, number of groups, number of aggregates computed, and number of bits used to encode values. Then, we evaluate the performance of BIPie in the context of TPC-H Query 1 and compare with previously published results.

All the tests were conducted on a computer with a single Intel Core i7-6700 3.4 GHz CPU and 32GB of RAM. The processor has four hardware cores and eight hardware threads. The evaluation of performance of individual operations was done outside of MemSQL engine using VectorToolbox library directly. The evaluation of query performance is done using a modified MemSQL engine. In every experiment, we utilize all available hardware threads simultaneously. This captures the effects related to the sharing of hardware resources, both between collocated hardware threads and between different physical cores. All of the input data is always in memory. We use inputs of at least hundred of millions of rows to make sure that the input does not fit into last level CPU cache. This ensures that we include the effects of main memory scan into

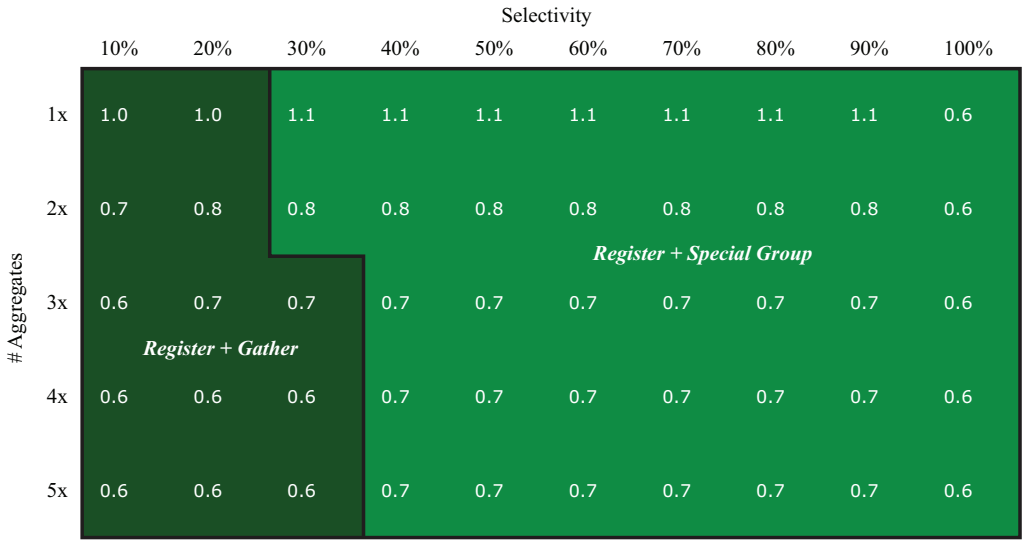


Figure 8: Performance of aggregation techniques, 8 groups, 7 bits encoding

our evaluation. We always run the same experiment ten times, and report the median of these ten runs.

As a unit of time measurement, we always use the elapsed CPU cycles per physical core, per input row, per computed sum aggregate. We further denote this as cycles/row/sum, except for the operations that do not compute aggregates and for SQL queries, for which cases we skip division by number of aggregates and use cycles/row. We find this unit gives a good intuitive understanding of the overall cost of an operation.

There are few more reasons why we believe reporting the CPU cycles per physical core per row is more useful than reporting the actual time. To a large degree, clock cycles abstract away some aspects of the hardware, such as the clock frequency or number of cores. Using clock cycles allows estimating what could be expected on a different machine with the CPU from the same family. Using CPU cycles per physical core per row also abstracts away the size of the input data in terms of number of processed rows. Division by number of computed sums helps in turn to observe how each additional aggregate added to a query becomes cheaper to compute in relation to previous ones. This is especially relevant to demonstrate the effect of the optimizations targeting the computation of multiple aggregates.

6.1 Selection Methods Comparison

Intuitively, the gather method is good for low selectivities, the compact method for medium selectivities, and special group for selectivities close to 1.0.

Figure 7 illustrates the performance measurements of selection with bit unpacking for two methods: compacting and gather. Graphs have been plotted for different bit widths of packed values (4, 7, 14 and 21). For each bit width, there is a fixed filter selectivity beyond which compacting starts to outperform gather. The graph shows gather performance to the left of this point and compacting to the right of this point, so only the best of two methods is shown for each selectivity and bit width. For example, compaction selection in 4 bits encoding outperforms the gather selection at a 2% selectivity. In the case of 21 bit encoding, gather outperforms compaction for selectivities below 38%. Performance numbers are expressed in CPU cycles per row.

6.2 Comparison of all combinations of Selection and Aggregation in BIPie

In this section, we evaluate the combined performance of selection and aggregation. We pair each of the three SIMD selection methods described (gather, compaction, and special group) with each of the three SIMD aggregation methods (sort-based, in-register, and multi-aggregate). This gives us a total of nine variants. We evaluate and compare the performance of all nine combinations across a range of parameters: the number of groups, the bit width used in the encoding of the columns that are being aggregated, the selectivity of the filter, and the number of aggregates (sums in our experiment).

Figures 8, 9, and 10 illustrate the best performing strategy for all selectivities across a various number of aggregates. The three tables shown contain the measurements repeated for three different numbers of groups and column encodings: 7 bit encoding with 8 groups (Figure 8), 14 bit encoding with 12 groups (Figure 9) and 28 bit encoding with 32 groups (Figure 10). In each cell, we determine the winning strategy and indicate the cycles/row/sum that it achieves. Groups of adjacent cells sharing the same best strategy have the same color. Each group is labeled with the name of the grouping strategy ('Sort' for sort-based, 'Register' for in-register, 'Multi' for multi-aggregate) followed by the name of the optimal selection strategy that was combined with the grouping strategy.

In this experiment, we use a very large input that is fetched from main memory to ensure the cost of main memory access is included. We use all available hardware threads. All input columns, group by columns, and aggregate columns are integers stored in a bit packed format.

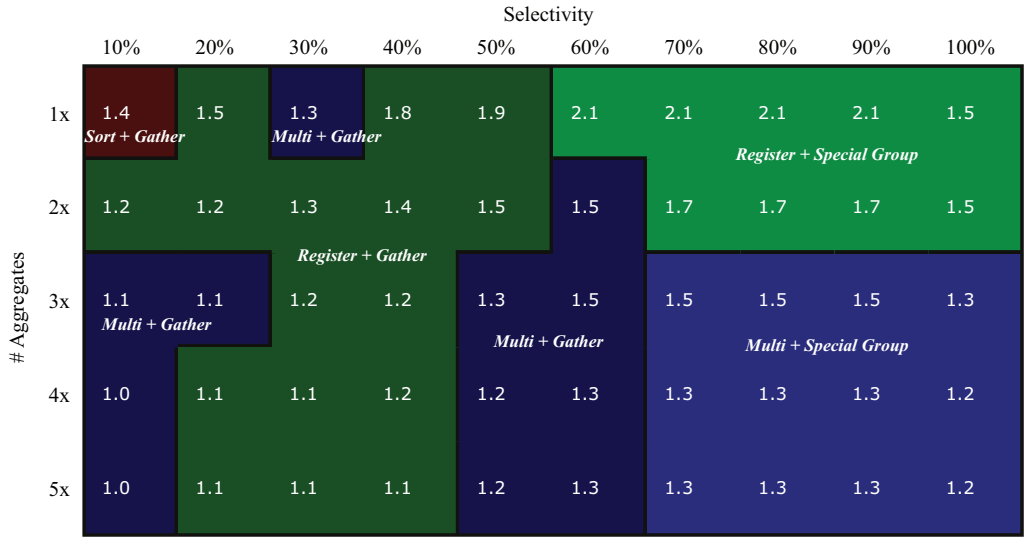


Figure 9: Performance of aggregation techniques, 12 groups, 14 bits encoding

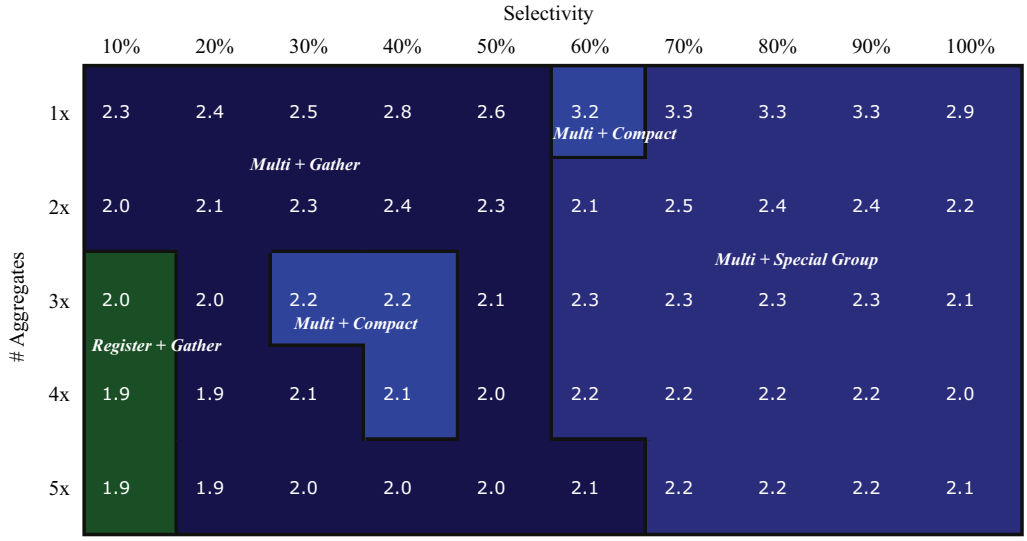


Figure 10: Performance of aggregation techniques, 32 groups, 28 bits encoding

As we increase the number of groups and number of bits used for encoding, from Figure 8, to Figure 9, and to Figure 10; we can notice a significant increase in the overall cost per sum. The increase is almost linear and is caused by a combination of the limitations of algorithms such as in-register aggregation, the higher costs of decoding values, and the higher memory bandwidth needed to bring input data into CPU cache.

Within each figure, the amortized cost per aggregate reduces with each new aggregate until it becomes stable. This is a result of two main factors. Processing of the filter byte vector and the group by column is a fixed cost that does not increase with the number of aggregates. There are also noticeable benefits from using specialized algorithms optimized for handling multiple aggregates.

The last column in Figure 8, 9, and 10 represents the scenario with no row filtering (100% selectivity). The constant portion of the total cost is reduced to the loading and decoding of the group by column. In Figure 8, there is no difference in the amortized cost per aggregate because the in-register aggregation strategy is used. By contrast, when the multi-aggregate method is used (Figure 10), there is a significant difference in the amortized cost per aggregate.

Next we analyze how the costs change with the selectivity. Interestingly, in most of the cases, there is not much change in the cost when varying the selectivity. In the first row in Figure 8, processing of all rows turns out to be even cheaper than selecting and aggregating just 10 percent of them. The latter case needs to handle the extra operation of loading the selection byte vector from

Engine	Scale Factor	#Rows (Approximate)	#Cores	Clock	Time [s]	Clocks/Row	Date published
EXASol 5.0 [5]	100	600000000	120	2.8	0.6	336	09/22/14
Vectorwise 3 [1]	100	600000000	16	2.9	1.3	100.5	04/15/14
SQL Server 2014 [4]	1000	6000000000	60	2.8	4.1	114.8	12/15/14
SQL Server 2016 [6]	10000	60000000000	96	2.2	13.2	46.5	11/28/16
Vectorwise 3 [2]	300	1800000000	16	2.9	3.8	98.0	05/10/13
Vectorwise 3 [3]	100	600000000	16	2.9	1.3	100.5	05/13/13
Hyper [16]	10	60000000	4	3.6	0.12	28.8	09/01/17
Voodoo [16]	10	60000000	4	3.6	0.162	38.9	09/01/17
CWI/Handwritten [11]	100	600000000	1	2.6	4	17.3	09/01/17
Hyper/Datablocks [12]	100	600000000	32	2.27	0.388	47.0	06/01/16
MemSQL/BIPie	100	600000000	4	3.4	0.381	8.6	

Table 5: Comparison of TPC-H Query 1 performance with previously published results

memory and converting it to an index selection vector to be used by the gather selection. Processing the selection vector has a cost proportional to the number of all input rows. As we increase the number of sums, this constant cost has smaller impact on per aggregate value. This explanation is not sufficient to fully understand Figure 8. We observe that experiments with low selectivity (10%) have a similar number of cycles/row/aggregates as experiments with 100% selectivity, even as the number of sums increase. We should notice that with a small number of bits per value, even if we selectively fetch a small portion of values, we are still likely to touch every cache line. Hence, the traffic from main memory to CPU cache will be very similar in both cases, with and without filter. The in-register aggregation method for a small number of groups is fast enough that the memory fetch dominates the cost.

In the results of this experiment, compacting selection is almost never a winner. However, it should be noted that this experiment does not consider more complex forms of aggregates, such as sums over arbitrary arithmetic expressions. The result of the experiment between compact and special group selection depends on the cost of post-filter processing of a row. As this cost grows, the compaction becomes a better choice. We can see in Figure 10 there are a few cells in the middle range of selectivities where compaction is the best approach. This indicates that both methods are very close in performance numbers for this combination of parameters.

For small bit width of encoded values in aggregate input columns and small number of groups, in-register group by aggregation is a clear winner. For higher bit widths and larger number of groups, multi-aggregate strategy is the best choice since its performance is not very sensitive to both of these parameters; unlike in-register method which linearly depends on both. For In-Register Aggregation, a smaller bit width means a higher degree of data level parallelism with SIMD, and smaller number of groups corresponds directly to smaller number of instructions.

6.3 Evaluation of BIPie in TPC-H Query 1

In this section, we put the techniques described in this paper to the test in an actual end-to-end processing scenario by running query 1 of the TPC-H benchmark using a modified MemSQL engine.

```
SELECT
 l_returnflag,
 l_linestatus,
 sum(l_quantity),
 sum(l_extendedprice),
 sum(l_extendedprice * (1 - l_discount)),
 sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)),
 avg(l_quantity),
 avg(l_extendedprice),
 avg(l_discount),
 count(*)
FROM
 lineitem
WHERE
 l_shipdate <=
  date '1998-12-01' - interval '90' day
GROUP BY
 l_returnflag,
 l_linestatus
ORDER BY
 l_returnflag,
 l_linestatus;
```

TPC-H Query 1 is a great example of a query that can benefit from the optimizations demonstrated in this paper. Query 1 is a single table scan containing a range filter on a date column selecting 98% of the rows. The query aggregates values into four groups, grouping on two string columns, both containing no more than three distinct values. All aggregates are either sums or averages with an extra count(*) aggregate. Two out of seven aggregates are computed over the results of a simple arithmetic expression involving multiple source columns of the table. The techniques presented in this paper also apply to multi-column group by, although generating the group id map for multi-column group by is out of the scope of this paper.

We use TPC-H at a scale factor 100 for our experiment. For this scale factor, the table that will be scanned contains close to 600 million rows, but the encoded columns that are used in query 1 can fit into the memory of the machine that we used. We sort and shard LINEITEM table on l_orderkey column, which is not used in this query, so we do not take advantage in any way of the order of rows in the table.

Now, let us discuss how the execution of query 1 is implemented. All operations described here work on a batch of rows (4096 rows in MemSQL) at a time. For each batch of rows, the filter is evaluated

using integer comparison using SIMD instructions, and its result is stored as a byte vector. Integer dictionary ids for both string group by columns are bit unpacked, stored in vectors of one byte elements, and then combined into a single integer values from the range 0 to 5. Even though the query outputs four groups, based on metadata we calculate that six groups are possible. Another seventh group is introduced by combining the byte vector of group indices achieved so far with the byte selection vector obtained from filter function (Special Group Selection). We evaluate the expressions using code generated at runtime. The code generated at runtime does not use SIMD. The In-Register Aggregation method is used for calculating the COUNT(*) aggregate. The Multi-Aggregate group by method is used to calculate all the sums. All five calculated sums can be updated for a single row in one load-add-store sequence of instructions, since the intermediate results fit into a 256-bit word.

In order to be able to compare the performance of Query 1 on different database management systems, we normalize various published results for various scale factors and represent them all in terms of cycles/row. We multiply execution time by nominal CPU clock frequency, multiply that by total number of physical cores on which the system is running, and finally divide by total number of rows in the table, which is proportional to scale factor. We acknowledge that this way of normalization is not a fair comparison, since it ignores many differences in the database setup and hardware configuration, such as different maximum column values for different scale factors, different memory speeds and differences coming from different CPU microarchitectures. Still, we believe that this is a meaningful overview of the progress on how fast query 1 can be executed. The authors of those publications put a special effort into configuring both hardware and database in an effort to achieve optimal results, applying their expertise in using respective products. Also, query 1 requires little synchronization coming from parallel processing, which limits the performance loss coming from high number of CPU cores. Furthermore, the query stresses CPU more heavily then the memory bandwidth. All selected publications are recent and utilize recent Intel CPUs.

BIPie achieves 8.6 CPU cycles/row. This processing cost is 2x lower than the recently published hand-written implementations by Gubner et al[11], and 3.3x lower than the fastest database engine implementation (Hyper).

7 RELATED WORK

There has been significant prior work on implementing database operators using SIMD. Zhou et al used SIMD to implement scan and join. A technique is also provided for aggregation without a group by clause [23]. More recently, work from Polychroniou et al. presented various techniques to make a number of query processing algorithms SIMD friendly. The paper introduced methods to optimize the use of a hash table using SIMD [17]. Willhalm et al. introduced techniques for SIMD-friendly decoding of values encoded in a columnstore and for the application of filters on encoded data using SIMD [22]. SQL Server in-memory extensions also implement a scan on encoded data, using SIMD decoding and SIMD filtering [13], as does DB2 BLU [18] and SAP HANA [22].

Voodoo uses static analysis to automatically introduce data-parallelism using SIMD [16]. Hyper compiles query using LLVM, but applies a tuple-at-a-time approach [15]. DataBlocks utilizes SIMD filtering on encoded data in the context of a hybrid transaction and analytics database using LLVM compilation [12].

GPU Databases [9] such as MapD [19], Kinetica, and BlazingDB aim to improve the performance of analytics by leveraging the massive parallelism of a GPU. In their widely-cited paper, Boral and DeWitt [8] made the case that using special-purpose hardware for database systems is not a good strategy, because general-purpose hardware advances at a faster pace than other hardware. Its unclear at this point whether GPUs are special-purpose or general-purpose. BIPie demonstrates that by optimizing algorithms for the modern CPU architectures, we can achieve performance similar to GPU databases, without limiting deployment options.

8 CONCLUSION

We introduce BIPie, a scan engine for ad-hoc analytical queries executed on compressed columnar data implemented in MemSQL. BIPie includes a collection of SIMD-friendly selection and aggregation algorithms, each optimal for a range of parameters. We introduce a new selection technique, the Special Group selection, optimized for higher selectivity when combined with aggregation. We present new SIMD-accelerated aggregation techniques. In-register aggregation is optimized for cases when the number of bits per value is small, and the number of groups is small. Multi-Aggregate takes advantage of SIMD to process multiple aggregates simultaneously. We evaluate the performance of those aggregation and selection techniques across a broad range of parameters. Then, we evaluate the end-to-end performance of those improved techniques in the context of the TPC-H query 1 benchmark. By normalizing query performance, we estimate that BIPie outperforms the best previously published result by 3.3x. We demonstrate that BIPie can compute query 1 in less than 9 CPU cycles per row.

9 ACKNOWLEDGEMENTS

We would like to thank the anonymous peer reviewer for their insightful feedback. We are grateful to the entire MemSQL engineering team, without whom this work would have been impossible. Specifically, Szu-Po Wang and PieGuy (David Stolp) for their valuable insight and feedback to BIPie. We would also like to thank Jilian Ryan, Kristi Lewandoski, and Amy Qiu.

REFERENCES

[1] 2013. TPC-H Lenovo ThinkServer RD630. http://web.archive.org/web/20170331123020/http://c970058.r58.cf2.rackcdn.com/individual_results/Lenovo/Lenovo-RD630-sf100-130510-ES.pdf. (2013). [Online; accessed 29-November-2017].

[2] 2013. TPC-H ThinkServer RD630. http://c970058.r58.cf2.rackcdn.com/individual_results/Lenovo/Lenovo-RD630-sf300-130510-ES.pdf. (2013). [Online; accessed 29-November-2017].

[3] 2013. TPC-H ThinkServer RD630. http://c970058.r58.cf2.rackcdn.com/individual_results/Lenovo/Lenovo-RD630-sf100-130510-ES.pdf. (2013). [Online; accessed 29-November-2017].

[4] 2014. TPC-H Cisco UCS C460 M4 Server. http://web.archive.org/web/20170421045119/http://c970058.r58.cf2.rackcdn.com/individual_results/Cisco/cisco~tpch~1000~cisco_ucs_c460_m4_server~es~2014-12-15~v02.pdf. (2014). [Online; accessed 29-November-2017].

[5] 2014. TPC-H Dell PowerEdgeR720xd. http://web.archive.org/web/20170421044648/http://c970058.r58.cf2.rackcdn.com/individual_results/Dell/dell~tpch~100~dell_poweredge_r720xd_using_exasolution_5.0~es~2014-09-23~v02.pdf. (2014). [Online; accessed 29-November-2017].

[6] 2016. TPC-H Cisco UCS C460 M4 Server. http://c970058.r58.cf2.rackcdn.com/individual_results/Cisco/cisco~tpch~10000~cisco_ucs_c460_m4_server~es~2016-11-28~v01.pdf. (2016). [Online; accessed 29-November-2017].

[7] Peter A Boncz, Marcin Zukowski, and Niels Nes. [n. d.]. MonetDB/X100: Hyper-Pipelining Query Execution.

[8] Haran Boral and David J. DeWitt. 1983. Database Machines: an Idea Whose Time has Passed? A Critique of the Future of Database Machines. (July 1983).

[9] Sebastian Breß, Max Heimel, Norbert Sigmund, Ladjel Bellatrech, and Gunter Aaake. 2014. GPU-accelerated Database Systems: Survey and Open Challenges. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV. Lecture Notes in Computer Science* 8920 (2014).

[10] Jack Chen, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvilli, and Michael Andrews. 2016. The MemSQL Query Optimizer: A Modern Optimizer for Real-time Analytics in a Distributed Database. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1401–1412. https://doi.org/10.14778/3007263.3007277

[11] Time Gubner and Peter Boncz. [n. d.]. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. *Proceedings of ADMS 2017.* ([n. d.]).

[12] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage Using Both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 311–326. https://doi.org/10.1145/2882903.2882925

[13] Per-Ake Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1740–1751. https://doi.org/10.14778/2824032.2824071

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

[15] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[16] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. *Proc. VLDB Endow.* 9, 14 (Oct. 2016), 1707–1718. https://doi.org/10.14778/3007328.3007336

[17] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[18] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1080–1091. https://doi.org/10.14778/2536222.2536233

[19] Christopher Root and Todd Mostak. 2016. MapD: A GPU-powered Big Data Analytics and Visualization Platform. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 73, 2 pages. https://doi.org/10.1145/2897839.2927468

[20] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. 2010. Fast Integer Compression Using SIMD Instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN '10)*. ACM, New York, NY, USA, 34–40. https://doi.org/10.1145/1869389.1869394

[21] A. Skidanov, A. J. Papito, and A. Prout. 2016. A column store engine for real-time streaming analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1287–1297. https://doi.org/10.1109/ICDE.2016.7498332

[22] Thomas Willhalm, Ismail Oukid, Ingo Mï£¡ller, and Franz Fï£¡rber. 2013. Vectorizing Database Column Scans with Complex Predicates. (08 2013).

[23] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/564691.564709