ROVEC: Runtime Optimization of Vectorized Expression Evaluation for Column Store

Meng Li, Zheyu Miao, Di Wu, Feifei Li, Sheng Wang, Wei Cao, Zhi Qiao
Yubin Ruan, Yukun Liang, Jimmy Yang, Haipeng Dai, Guihai Chen

Abstract—Due to the increasing demand for scalable and interactive data analytics, column stores have become the de-facto choice in many analytical databases. As a common and fundamental operation in column stores, expression evaluation has a remarkable effect on many queries. To speed up expression evaluation, vectorized techniques such as Single-Instruction-Multiple-Data (SIMD) instructions are widely used. However, there are few works concerning dedicated optimizations for SIMD-based expression evaluation for column stores. In this paper, we propose a runtime optimization framework named ROVEC that enables effective optimizations for SIMD-based expression evaluation. The key idea is to optimize logical expression at execution time, by leveraging lightweight compression and fine-grained statistics associated with the compressed data. ROVEC removes unnecessary type casting and finds the tightest type during evaluation, which maximizes the concurrent operands in SIMD instructions. ROVEC can be applied to many expression-evaluation-intensive operators (*e.g.*, table scan and theta join) for different data types (*e.g.*, numeric, time and string). To validate the effectiveness of ROVEC, we integrate it into a columnar database PolarDB-C. Our evaluation results show that ROVEC improves up to 125% (60% on average) throughput of table scan and up to 50% (30% on average) latency of theta join.

Index Terms—Expression Evaluation, SIMD, Column Database

1 Introduction

Analytical databases based on columnar storage (*e.g.*, Redshift [1], Snowflake [2], and Vectorwise [3]) have become the de-facto choice for enabling scalable and interactive analytics over a large amount of data, due to their excellent I/O efficiency. In a column store, data from different columns is separated into different physical files, while data from the same column is grouped as consecutive fix-sized blocks that are compressed into files. Such a storage layout allows faster evaluation on selected columns without loading irrelevant columns as in traditional row stores, catering to the rising demands of large-scale interactive queries and analysis. Furthermore, a block-level statistical summary (*e.g.*, min/max) is associated with each block, which can be used to further reduce I/O by bypassing evaluations on obviously unqualified or qualified blocks [4].

However, block-wise information has been poorly used regarding expression evaluation [4]. On one hand, at the SQL-layer, the block-wise information cannot be used by a SQL-layer optimizer since they usually work on column-wise statistics where block-wise information is transparent to them. On the other hand, at the executor-layer, the block-wise information is only used in the most straightforward way, *e.g.*, filtering out simple predicates evaluation on a block according to its min/max values [4]. Such poor utilization is due to the inevitable overhead of looking up to the block-level summaries, especially if not enough benefits are achieved. To make full use of the block-wise information, we observe a particular optimization opportunity: type reduction of expression evaluation, *i.e.*, packing data into smaller types with block-wise information to improve the parallelism of widely-used SIMD-based evaluation. Note that SIMD instructions are widely used for expression evaluation in column stores [5], [6], [7], by conducting the same operation on multiple values simultaneously. The benefits of type reduction can be further amplified by newer SIMD instructions (*e.g.*, AVX-512) since their (wider) registers accommodate more values at one time.

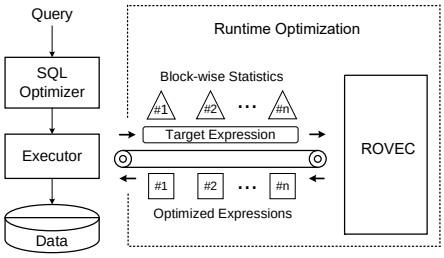


Fig. 1: Runtime Optimization

However, type reduction for general expressions is non-trivial due to overflow/underflow and block-by-block optimization overhead. Hence, it is no wonder that only some simple expressions, like constant predicates[1] [5] and boolean predicates[2] [8], [9], are supported. Another similar

- *Meng Li is with State Key Laboratory for Novel Software Technology, Nanjing University. Zheyu Miao is with Zhejiang University. E-mail: menson@smail.nju.edu.cn, zheyu.mzy@alibaba-inc.com*
- *Di Wu, Feifei Li, Sheng Wang, Wei Cao, Zhi Qiao, Yubin Ruan, Yukun Liang, and Jimmy Yang are with Alibaba. E-mail: {wd154004, lifeifei, sh.wang, mingsong.cw, george.qz, yubin.ryb, liangyukun.lyk, xinjun.y}@alibaba-inc.com*
- *Haipeng Dai and Guihai Chen are with State Key Laboratory for Novel Software Technology, Nanjing University. E-mail: {haipengdai,gchen}@nju.edu.cn*
- *Feifei Li and Guihai Chen are the corresponding authors.*

1. The predicates between a column field and a constant.
2. The predicates only containing comparisons operations, and excluding arithmetic and other operations like $+$ or IN.

type optimization from Vectorwise [10] is also discussed in Section 6. Therefore, how to support type reduction for general (*e.g.*, non-constant or non-boolean) expressions remains unsolved. To address this problem, we propose ROVEC (**R**un-time **O**ptimization of **V**ectorized **E**xpression evaluation for **C**olumn store), which is embedded in execution engines and is a complement to traditional SQL-layer optimizers. As shown in Figure 1, ROVEC takes an expression and a block summary as input and outputs an executable expression for each block individually. Its key idea is to postpone the optimizations to execution time, in order to utilize fine-grained block-wise statistics for type reduction of (input/intermediate) data during evaluation. In ROVEC, several optimization rules are carefully selected and laid out to maximize the chances of type reduction with minimized overhead. Particularly, considering the wide use of lightweight compression schemes [11], [12], [13] in column stores, ROVEC unfolds the decompression process of a compressed block as a subexpression, which creates further opportunities for type reduction. Also note that some rules are already implemented in compilers like LLVM [14] and GCC [15], which however, cannot be borrowed directly here, due to block-by-block compiling latency. Finally, we emphasize that ROVEC is not a column store but just an expression evaluation framework, which can interact with column stores through a concise API and thus does not require major architectural changes of column stores.

Challenges. There are four major challenges for the design and implementation of the ROVEC framework.

The first challenge is how to choose proper compression schemes that bring in opportunities for runtime optimization. To address this challenge, we have defined a set of element-addressable (EA) schemes, which are a subset of widely-used lightweight compression schemes that can decompress each compressed value independently. These schemes are applicable to various data types, including numeric, time, and string. Besides, in ROVEC, both EA schemes and non-element-addressable (NEA) schemes can be applied in combination, *i.e.*, applying EA on raw data and then NEA on EA-compressed data. During evaluation, only the NEA layer is decompressed and provided to ROVEC.

The second challenge is how to avoid underflow/overflow during type reduction. To address this challenge, a lightweight type safe reduction algorithm is proposed in Algorithm 1, which firstly estimates (or calculates) the value range of each node of the input expression, and then finds the tightest type for each expression node.

The third challenge is how to mitigate the overheads from block-wise optimization. ROVEC optimizes expression evaluation for each block individually, which inevitably introduces overheads that may outweigh the benefits if not carefully handled. To address this challenge, ROVEC only relies on tiny block-wise summaries that can be persistently cached in memory, making lookups to them highly efficient. Besides, ROVEC only needs to traverse the expression tree a few times for each block, which is negligible (*i.e.*, $< 9\%$) compared to the subsequent computation on the block.

The fourth challenge is how to make ROVEC applicable to different operators. To that end, ROVEC is designed as a standalone component, such that all operators can interact with it seamlessly. To obtain optimized expressions, each operator only needs to provide the logical expression along with corresponding block-wise statistics. Hence, the support for different operators can be implemented independently. The expression evaluation intensive operators like table scan and theta join [16] have been supported in ROVEC.

Contributions. We make contributions to the runtime optimization of expression evaluation in three folds. Firstly, we propose a runtime optimization framework ROVEC for expression evaluation, which adopts several novel runtime optimizations to better exploit type reduction under SIMD-based execution. Secondly, to the best of our knowledge, we are the first to generalize the concept of type reduction under lightweight compression schemes and combine it with SIMD-based execution, which is shown to help achieve significant performance gain in various workloads. Thirdly, we conduct extensive evaluations to verify the effectiveness of ROVEC on different data sets, by plugging it into a column database PolarDB-C, (the columnar version of PolarDB, [17], [18], a cloud-native database developed within Alibaba for Alibaba Cloud). Our results show that, compared with PolarDB-C, ROVEC increases up to 125% (60% on average) throughput of table scan and reduces up to 50% (30% on average) latency of theta join.

2 Background

Column-block storage format. In such storage format, consecutive records from a column are grouped into blocks, which are then compressed and stored into files. Besides, each block is associated with a block summary (containing Min, Max, Sum, compression schemes and *etc.*). Due to data distribution variances, blocks from the same column may yield different compression ratios and types.

Element-addressable scheme. Suppose C is a compression scheme, $\widehat{C}$ is its decompression scheme, and $\{s_1, \cdots, s_n\}$ is a data block . C is element-addressable if compressed data $C(s_i)$ can be decoded independently, *i.e.*, $s_i = \widehat{C}(C(s_i))$. Example schemes in ROVEC are as below:

(1) Single-value scheme. Single-value scheme applies to the case that all values within a block are identical. This scheme is simple but useful in many cases, *e.g.*, date field of stock data within the same day. This scheme is applicable to both numeric and non-numeric data types.

(2) Dictionary scheme. Dictionary compression is a widely-used scheme, which maps each distinct value from long-length records to a short-length index. Then, we replace each value with its corresponding index in the compressed block. Dictionary scheme is used for string type.

(3) FOR scheme. FOR (*i.e.*, frame of reference) [19] compresses a block by storing the delta between each value and the minimum value within the block. Since deltas are usually much smaller than original values, they can be loaded into smaller types and thus reduce the total volume. FOR scheme is used for numeric values.

(4) GCD scheme. GCD (*i.e.*, greatest common divisor) scheme compresses a block by replacing each original value with the one divided by the greatest common divisor of all values within the block. Specifically, suppose s_{GCD} denotes GCD of the block, then the block is compressed to $\{\frac{s_1}{s_{\text{GCD}}}, \cdots, \frac{s_n}{s_{\text{GCD}}}\}$. GCD is applicable to numerical values.

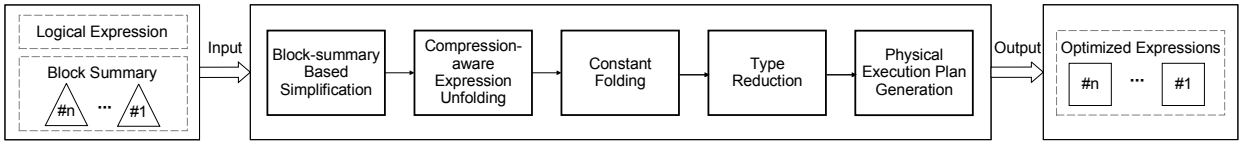


Fig. 2: Overview of ROVEC Framework

3 Design of ROVEC

In this section, we introduce the critical designs of ROVEC as shown in Figure 2. We start with the insight behind our framework design and the key optimization rule leveraged by it named **type reduction**. To better demonstrate each component, we provide a running example of expression evaluation, and then the full elaboration on each step.

Generally, the optimization opportunity of ROVEC originates from the block summaries, which cannot be utilized by a conventional SQL-layer optimizer since they usually work on column-wise statistics. To make full use of the block-wise information, a particular optimization opportunity is observed: type reduction. Evaluating expression with type reduction can be beneficial in three ways: (1) coherent evaluation procedure that combines decompression and expression together seamlessly; (2) avoiding the overhead of transferring (between cache, memory or storage) data of wider types; (3) faster SIMD-based evaluation (with smaller data types). However, the support of type reduction of general expressions is non-trivial due to underflow/overflow and optimization overhead. To address the first problem, we propose a lightweight algorithm, which supports first estimating the value ranges of each expression node and then reducing the type of each node to the tightest one. To address the second problem, in ROVEC, a collection of lightweight optimization rules are carefully selected and laid out to maximize the chances of type reduction while the overall optimization overhead is maintained to be negligible. Particularly, the second rule, *i.e.*, unfolding the lightweight schemes as subexpressions, further empowers type reduction optimization by enabling operations on compressed data directly. As for the other rules, they aim at removing redundant expression evaluation regarding constants. The removed evaluation makes the expressions provided to the type reduction module as simple and tidy as possible, which correspondingly reduces the overhead of the type-reduction procedure. During runtime, ROVEC traverses the expression and applies the associated rules as shown in Figure 2.

Compared with previous works, our main novelty lies in building a runtime (*i.e.*, execution time) lightweight optimizer that aims to reduce the evaluation overhead block by block. Note that some optimization rules (*e.g.*, constant folding) in ROVEC may not be new but are used in a significantly different way compared with previous works in the SQL layer. In the first fold, we are the first to combine these optimizations together and bring them into the execution layers with negligible overhead. In the second fold, if directly applying our optimizations rules in the SQL layer (based on the coarse column-level statistics), only limited or even no performance gain will be achieved; in contrast, based on the fine-grained block-wise statistics, a significant performance gain can be achieved. For example, considering

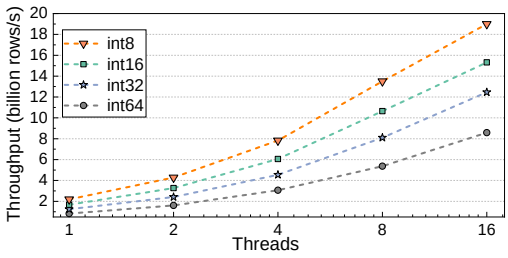


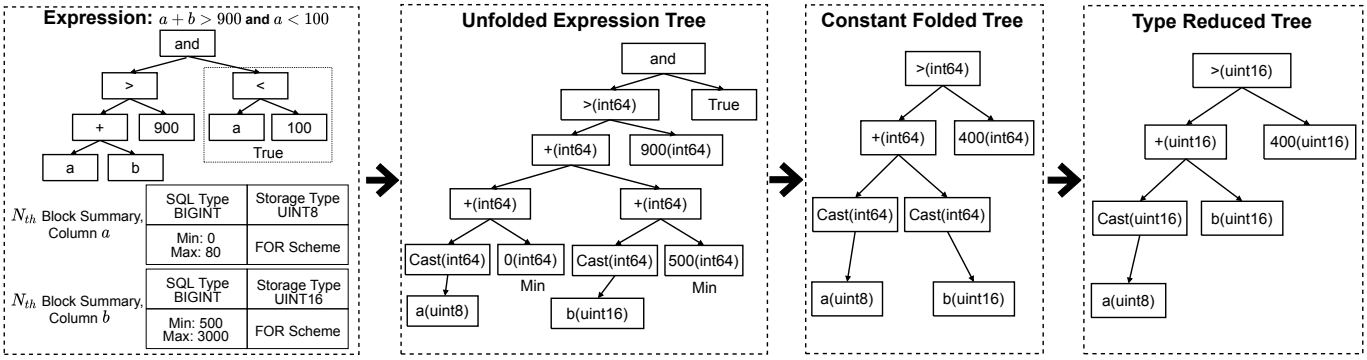
Fig. 3: Micro Bench of Type Reduction

that the data values from a whole column usually cover a broad range, reducing the evaluation type of the data from a whole column is usually infeasible, which, however, is feasible within our fine-grained block-wise optimization framework. In the third fold, instead of a straightforward combination of existing optimization rules, the applied optimizations in our work are elaborately selected and carefully ordered to maximize the chances of our key optimization rule, *i.e.*, type reduction.

3.1 Key Optimization: Type Reduction

Usually, an expression is evaluated with its input data types, which are derived from the user-defined table scheme. However, accessing compressed data inevitably incurs data type promotion *i.e.*, converting from small compressed type to the larger defined type in the table schema. Recall that evaluations of predicates like $(>)$ can be easily accelerated by SIMD instructions. In addition, since SIMD operates on fixed-length registers, using smaller types allows for more values processed by one single instruction. In principle, up to $2\times$ throughput improvement can be achieved when the value width is reduced by half. We conduct a microbenchmark to demonstrate the end-to-end performance of SIMD instructions (*i.e.*, AVX512) for equality predicate evaluation of different integer types, as depicted in Figure 3. The results show that $25\% \sim 40\%$ throughput improvement is observed when the type is reduced by one level (*i.e.*, width reduced by half). We call it *type reduction* throughout this paper.

The most direct opportunity for type reduction lies in the block summary. Unlike data types defined in table schema, which have to cover the possible maximum value in the entire column, fine-grained statistics in block summary can reflect distributions of contained block-wise values more accurately. For example, in expression $COL1 > COL2$, if both fields (*e.g.*, in $INT64$ type) can be covered by a smaller type (*e.g.*, $INT16$), predicate $(>)$ can also be evaluated on this smaller type. Moreover, a more aggressive type reduction can be achieved if the evaluation can be conducted directly on compressed smaller types, which, however, needs to be based on proper (*i.e.*, element-addressable) compression schemes. To achieve that, we need to unfold the decompression process of element-addressable compression schemes as subexpressions and embed it into the logical expression.



(a) Simplification (b) Expression Unfolding (c) Constant Folding (d) Type Reduction

Fig. 4: Running Example

Specifically, considering that element-addressable schemes enable operations on per-element granularity, their decompression can be unfolded into subexpressions as follows:

- *Single-value can be represented as a single constant;*
- *Dictionary can be represented as a dictionary lookup operator;*
- *FOR can be represented as a pair of $Cast$ and Add operators;*
- *GCD can be represented as a pair of $Cast$ and Mul operators.*

With the unfolded decompression above, type reduction optimization will not only be conducted within the unfolded subexpression but will also be propagated to other (comparison or arithmetic) operations in the logical expression. For example, the comparison between string values compressed by Dictionary scheme can be converted into the comparison between (length-reduced in bytes) dictionary indexes. Besides, if $COL1$ and $COL2$ are compressed with FOR scheme, the above expression can be rewritten as $COL1' > COL2' + (Min2 - Min1)$, where $COL1'$ and $COL2'$ are the compressed values, and $Min1$ and $Min2$ are minimum values in the corresponding block summaries. Apart from the comparison operators, other arithmetic operators (such as $+, -, *, /$) can also be optimized in a similar fashion. In summary, with type reduction, we can significantly boost the expression evaluation, which could be further boosted if the data are compressed into smaller types.

3.2 A Running Example

Before diving into the details, we first give a running example to illustrate the intention of each rule. In Figure 4(a), the example expression is $(a+b > 900\ and\ a < 100)$, where a and b are columns of $BIGINT$ type and compressed with FOR scheme into storage type of $UINT8$ and $UINT16$. Besides, an example block summary (denoted by $\widehat{a_N}$) of block a_N (the N_{th} block of column a) is also shown in the figure, as well as the block summary $\widehat{b_N}$ of block b_N.

Block-summary based simplification. This stage aims at identifying the potential constants in the given expression with the statistics from the block summary. As shown in Figure 4(a), in the given expression $(a+b > 900\ and\ a < 100)$, the subexpression, *i.e.*, $a < 100$, might result in a constant according to the block summary $\widehat{a_N}$. Specifically, as the max field (*i.e.*, $Max(\widehat{a_N})$) is smaller than 100, this subexpression ($a < 100$) is evaluated to be $True$ and thus leads to the simplified expression $(a + b > 900\ and\ True)$.

Compression-aware expression unfolding. This stage aims at, for each block, unfolding the compression schemes into subexpressions to be further optimized in the later stages. As shown in Figure 4(b), ROVEC unfolds column a as $CastToInt64(a_N) + Min(\widehat{a_N})$, and column b as $CastToInt64(b_N) + Min(\widehat{b_N})$ similarly.

Constant folding. This stage aims at eliminating redundant evaluation regarding constants. For example, constants (*e.g.*, $Min(a_N)$ and $Min(b_N)$) have been explicitly embedded into the expression during the unfolding stage. Now, we can swap the constants to the right side of the expression, resulting the new expression $CastToInt64(a_N) + CastToInt64(b_N) > 900 - (Min(\widehat{a_N}) - Min(\widehat{b_N}))$. Therefore, the costly arithmetic evaluation, *e.g.*, $CastToInt64(a_N) + Min(\widehat{a_N})$, is eliminated as is shown in Figure 4(c).

Type reduction. In Figure 4(c), without type-reduction, block a_N and b_N need to be promoted to $INT64$ during evaluation, which is time-consuming and unfriendly to SIMD. On the contrary, ROVEC estimates the value ranges of the expression rooted by node $(>)$ recursively. Since the max value of a_N and b_N are 80 $(80 - 0)$ and 2500 $(3000 - 500)$, $a_N + b_N$ is no larger than 2580 and can be covered by type $UINT16$. A feasible type reduction is shown in Figure 4(d).

Physical expression generation. To generate the executable expression (*i.e.*, the physical plan), we assign a physical implementation to each node in the expression tree, and convert recursive tree-like execution to sequential queue-based execution via DFS (depth-first search).

3.3 Optimization Strategy in ROVEC

In this subsection, we elaborate on the optimization rules of ROVEC in Figure 2. Note that the capability of ROVEC is not limited to the rules listed, and additional rules can be appended on-demand to handle new compression schemes, statistical summary (*e.g.*, bloom filter), and patterns (*e.g.*, SQL functions).

3.3.1 *Summary-based Simplification*

Summary-based simplification is the first stage, which utilizes fine-grained statistics from the block summary to reduce computation within the given expression at runtime. Recall that a block summary is associated with each data block and contains block-wise statistics. Especially, the following statistics are extensively utilized: min/max, SQL

data type, storage data type, and compression schemes. To simplify an expression, ROVEC first traverses the input expression via DFS to identify optimizable patterns. An optimizable pattern is an expression sub-tree that can be potentially evaluated as a constant with the help of block summary ($True$ or $False$, *e.g.*, $a < 100$). If any optimizable pattern is recognized as a constant, the corresponding sub-tree will be replaced by the constant. In this way, the conventional element-wise evaluation is replaced by block-wise evaluation, which only needs to be done once per block.

3.3.2 Compression-aware Expression Unfolding

Expression unfolding is an important stage that opens the gate for effective type reduction. As mentioned, each column is compressed in the unit of blocks and needs to be decompressed when an expression involves the evaluation over this column. However, since the compressed values are stored in smaller data types, decompressing them not only incurs extra computation overhead but also misses the opportunity to reduce the evaluation data type. To address this issue, we propose to unfold the decompression process for each data block as a subexpression according to the schemes applied to the block.

During the unfolding process, the strategies for different schemes are not the same. For the GCD scheme, we need to unfold the greatest common divisor of the block as a constant in the expression. For the Single-value scheme, unfolding the compressed block is equivalent to replacing it with the single constant from the block summary. As for Dictionary scheme, we reinterpret a data item into the corresponding dictionary index. Note that ROVEC does not restrict a block to be compressed by only one scheme. If a block is compressed by multiple schemes in cascade, ROVEC unfolds the applied schemes one by one according to their applied order. By doing so, instead of decompressing values directly, ROVEC unfolds the decompression process as a subexpression, which can be optimized in the follow-up stages.

3.3.3 Constant Folding

Constant Folding stage takes the output from the previous stage as input and aims at removing redundant evaluations. As discussed in Section 3.3.1, constants might be generated and embedded in the expression. Such constants can be further propagated to simplify the unfolded expression. For example, the expression $(a+b > 900\ and\ True)$ is equivalent to $(a+b > 900)$ based on the conjunction simplification rule.

We also discover other opportunities that can further prune out computation regarding constants. A typical case is called *constant swap*, as shown in Figure 4(c). Note that overflow/underflow issues need not be worried during the constant swap, since value ranges of intermediate evaluations can be inferred from Min/Max values of the block summaries, and if needed, properly capped. In summary, constant folding helps remove redundant evaluations by merging intermediate evaluations that only involve constants, which is also applicable for Single-value, Dictionary, and GCD schemes.

Algorithm 1: ReduceType

```
Data: expression node n
Result: type reduced expression
1 for c in the child nodes of n do
2     ReduceType(c);
3 {T_out, T_in} = DoEstimate(n); /* estimated tightest
      type                                              */
4 if n.T_in > T_in then
5     if n is the root node and n.T_out < T_out then
6         add a Cast node (from n.T_out to T_out) as n's
            parent;
7     n.T_out = T_out;
8     DoReduce(n, T_in);
```

3.3.4 Type Reduction

The type reduction stage takes the output from previous steps as input and outputs a type-reduced expression. It is a novel optimization mechanism proposed by ROVEC, which enables runtime acceleration of expression evaluation. This optimization opportunity originates from two aspects. First, compression schemes may reduce user-defined SQL types into smaller data types. Second, the parallelism of SIMD-based evaluation benefits from smaller data types, *i.e.*, evaluating more operands (*i.e.*, values) as the size of each operand decreases. For example, in the Intel AVX-512 instruction set, a SIMD instruction can handle 64 bytes of operands. It can only process 8 items if they are 64-bit integers, but up to 64 items if they are 8-bit integers. Type reduction packs items into a type as small as possible so that a set of items can be evaluated with fewer instructions.

However, unlike normal type assignments in SQL optimizer, one important consideration in ROVEC is to avoid introducing new overflow/underflow issues while reducing evaluation type. To solve this problem, ROVEC first estimates the value range of each intermediate expression node, and then reduces its type to the tightest one that covers the estimated range. This estimation is feasible since block-wise statistics reflect the value range within each block.

The main procedure of type reduction is shown in Algorithm 1. It traverses the input expression tree via DFS to identify the tightest (input/output) types for each node, *i.e.*, each node firstly estimates its range (Step 3, Algorithm 1), and secondly reduces its type (Step $4-8$, Algorithm 1).

In the first phase, each node estimates its output range according to its own node type and the output ranges of its direct child nodes by invoking Algorithm 2. For example, the constant and field nodes derive their output ranges from the block summaries (Step $1-3$ and $4-5$ in Algorithm 2, respectively). As for the scalar functions, their output ranges are derived from the output ranges of direct child nodes, *e.g.*, function Add in Step $8-12$ of Algorithm 2. However, if estimation is not feasible, the range of each node's original type is returned (Step $14-16$, Algorithm 2). With the estimated range, the type reduction process is triggered for each node to chooses its tightest type.

With the estimated type, the reduction process is triggered for each node when its estimated type is smaller than the original type (Step $4-7$, Algorithm 1), and then Algorithm 3 is invoked to reduce the type of the direct child nodes (Step 8, Algorithm 1). Specifically, if a child node is a

Algorithm 2: DoEstimate

Data: expression node n
Result: estimated tightest output type t_{out} and input type t_{in}

1 **if** *$n.op$ is Const* **then**
2 $n.max = n.val,\ n.min = n.val$;
3 $t_{out} = t_{in}$ = the tightest type covering the range;
else if *$n.op$ is Field* **then**
4 derive $n.min$ and $n.max$ from block summary;
5 $t_{out} = t_{in}$ = the tightest type covering the range;
else if *$n.op$ is Predicate* **then**
6 $n.min = 0,\ n.max = 1,\ t_{out} = boolean$;
7 t_{in} = the tightest type covering children's output type;
else if *$n.op$ is Add* **then**
8 $n.max = n.children[0].max + n.children[1].max$;
9 $n.min = n.children[0].min + n.children[1].min$;
10 t_{out} = the tightest type covering the range of n;
11 t_{child} = the tightest type covering children's output type;
12 $t_{out} = t_{in}$ = the larger type between t_r and t_{child} ;
else if *other specific operations* **then**
13 `/* operation-specific estimations */`
else if *Unsupported estimation* **then**
14 $t_{out} = n.T_{out},\ t_{in} = n.T_{in}$;
15 $n.min$ = min value in t_{out}, $n.max$ = max value in t_{in} ;
16 **return** $\{t_{out},\ t_{in}\}$;

Algorithm 3: DoReduce

Data: expression node n, new input type T_{in}
Result: None

1 $n.T_{in} = T_{in}$;
for *c in child nodes of n* **do**
2 **if** *$c.op$ is $Cast$* **then**
if *$c.T_{in}$ is the same as T_{in}* **then**
3 remove the $Cast$ node c;
else if *$c.T_{out} > T_{out}$* **then**
4 $c.T_{out} = T_{in}$; `/* Cast to a smaller type */`
else if *$c.op$ is $Const$* **then**
5 $c.T_{out} = c.T_{in} = T_{in}$;
else if *$c.op$ is Scalar Function and $c.T_{out} < T_{in}$* **then**
6 add a $Cast$ node (from $c.T_{out}$ to T_{in}) as n's parent;

type casting ($Cast$) node, its output type is reduced to the input type T_{in} as Step $2-4$ of Algorithm 3 shows. In the best case, if the reduced output type is equal to its original input type, the $Cast$ node can be entirely removed (Step 3, Algorithm 2). Constant nodes can be modified directly, as they do not involve any calculation (Step 5, Algorithm 3). As for the child scalar function nodes, their output types remain unchanged since they are already reduced to the tightest type (Step 7, Algorithm 1). However, in case that the input type is larger than the output type of a child scalar function node, a new $Cast$ node will be added as Step 6 of Algorithm 3 shows. In such scenarios, adding a new type casting node to the current node is equivalent to postponing the type casting of child nodes, which could be further optimized in the later steps.

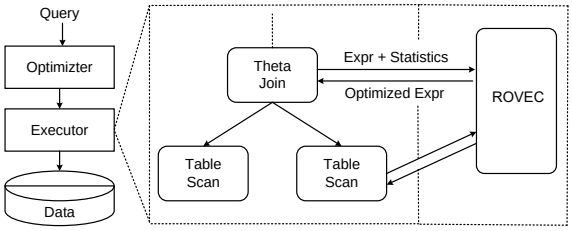


Fig. 5: System Implementation

4 System Implementation

In this section, we demonstrate how ROVEC can be implemented on top of an existing database system, taking PolarDB-C as an example. It is noteworthy that the adoption of ROVEC does not require major architectural changes.

4.1 Overview

As shown in Figure 5, ROVEC is implemented as a standalone component that all execution operators can interact with it seamlessly through a concise API. For each interaction, ROVEC receives the logical expression along with corresponding block summaries as input and outputs a block-wise optimized expression. The execution framework of PolarDB-C follows the traditional volcano architecture, where rows are pulled from child operators. Each pull action fetches a data chunk (*i.e.*, thousands of rows from the same block) from a child operator. Besides, each pulled chunk is attached with an extra *block ID*, which can be used to retrieve the block summary required by ROVEC.

4.2 Operator Support

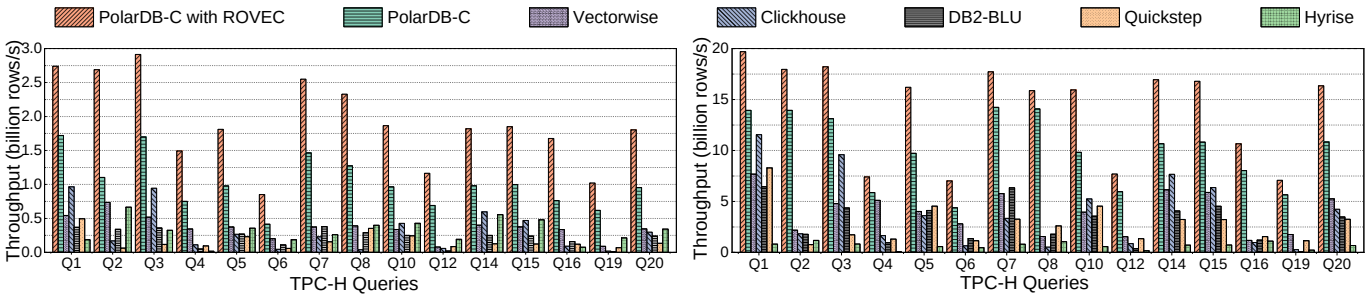
Table scan. Table scan operator scans the underlying columnar table to extract the IDs of qualified rows given a predicate. During a table scan, ROVEC is triggered for each row group (a group of rows from the same block), and it generates an optimized predicate to enable fast evaluation of the current block using SIMD.

Theta join. There are different ways to implement a theta join. The most generic approach, which can handle arbitrary theta-join conditions, is the (block) nested loop join. Nested loop join is a computation-intensive operator but with the best coverage in terms of supported join types. It fetches data chunks from both child nodes and compares all pairs of rows via a nested loop. For each pair of chunks (*i.e.*, one from the left child and the other one from the right child), ROVEC is triggered to generate an optimized join predicate.

Other operators. Note that with the support for table scan and generic theta join, ROVEC can support standard SPJ queries (as well as Group By, since Group By relies on similar semantics as projection). How to extend ROVEC to support nested query blocks with cross-block optimization is an interesting future work that we will explore.

4.3 Non-Element-Addressable Scheme

In this subsection, we illustrate how to support Non-Element-Addressable (NEA) schemes, which refers to compression schemes other than the Element-Addressable (EA) schemes defined in Section 2. Example NEA schemes include GZip [20], Run-length-encoding [11].



(a) Thread Count = 1

(b) Thread Count = 32

Fig. 6: Overall Table Scan Performance on TPC-H

From a system's perspective, ROVEC does not exclude other compression schemes. Specifically, both EA and NEA can be applied in ROVEC in a stacked way, *i.e.*, apply EA on raw data first and then NEA on EA-compressed data. Correspondingly, during evaluation time, only the NEA layer is decompressed, so that high compression ratio and runtime optimization opportunities can be both sustained.

Our runtime optimizer ROVEC, as a plugin embedded in the query executor, does not change the architecture of the database system like the underlying storage engine in which NEA is applied. Therefore, whether NEA is used or not in the underlying storage engine is transparent to ROVEC. Moreover, applying EA will not affect the effectiveness of subsequent NEA. In other words, even if NEA is applied, the performance gap between ROVEC and the comparison system (PolarDB-C) is generally unchanged since their performance will increase/decrease simultaneously.

5 Experimental Evaluation

To validate the effectiveness of ROVEC, we conduct an experimental evaluation for ROVEC by integrating it into a columnar database PolarDB-C. We extensively evaluate ROVEC with queries extracted from TPC-H [21] and a real-world a recommendation system workload. The experiments include evaluations of two expression-intensive different operators (*i.e.*, table scan and theta join), together with comparisons with other systems. Note that considering that there are other operators (other than table scan and theta join) that dominates the execution time of queries in TPC-H, we extract the table scan and theta join queries from TPC-H to directly demonstrate the performance improvement brought by ROVEC.

5.1 Experimental Setup

Hardware. The experiments are conducted on a server with 32 cores based on Intel Xeon Platinum 8163 CPU (@2.50GHz), 128GB memory, and 1.0TB SSD.

Metric. To achieve a head-to-head comparison, we use *throughput* (*i.e.*, the number of rows scanned per second) as the metric to measure the performance of the table scan operator. For the theta join operator, we use *latency* (*i.e.*, end-to-end response time) as the metric. Higher throughput or lower latency indicates better performance.

Baseline systems. In the experiments, we compare ROVEC with the original PolarDB-C, as well as other SIMD-based column databases including Vectorwise (or Actian Vector) [22], ClickHouse [23], DB2 with BLU [9] (DB2-BLU for short), QuickStep [24], and Hyrise [25]. Vectorwise is a column database deeply integrated with many query processing innovations such as vectorized execution model [26]. Particularly, Vectorwise extensively exploits performance-critical features of modern CPUs like super-scalar execution and SIMD instructions. ClickHouse is a linearly scalable and hardware efficient column store [23], which utilizes SIMD instructions but of an old version (*i.e.*, AVX-256). DB2-BLU is another column store with an innovated compression scheme that enables multiple compressed values packed into a register to improve the SIMD-based parallelism of evaluating operations such as predicates and joins. As for Quickstep, it is designed to be scaling-up and equipped with a vast body of techniques for organizing data, optimizing, scheduling and executing queries. Hyrise is a main memory database, which automatically partitions tables into vertical partitions of varying widths according to column access pattern. PolarDB-C is implemented with the newest SIMD instructions (*i.e.*, AVX-512) at all layers, ranging from query processing to data compression/decompression. Considering that the latest AVX-512 instructions are not supported by all the systems above, PolarDB-C is used here as a baseline for AVX-512-based systems. The comparison between PolarDB-C and other systems reflects the effectiveness of the AVX-512 instructions while the comparison between PolarDB-C and ROVEC reflects the performance gain from the proposed optimizations.

TPC-H dataset. We use 100GB TPC-H to evaluate the table scan operator. For theta join operator, as it is highly computation-intensive, we use 1GB TPC-H. Note that TPC-H queries contain many different operators. To better demonstrate the performance of the table scan operator, we extract only those queries (or sub-queries) that require expression evaluation, as listed in Table 1. These queries are selected in the principle of covering as many types of queries as possible. Considering that there is no theta join in TPC-H, two theta join queries are synthetically constructed as shown in Table 3, which is inspired by [27].

RO dataset. To further validate the effectiveness of ROVEC in real-world applications, another dataset named RO (Recommendation Online dataset) is used, which is about 192GB and collected from the production environment of an online recommendation system. As RO is in the production environment, it is disallowed to be imported into systems (like Vectorwise) unavailable in the environment. Hence, only the results of ROVEC and PolarDB-C are

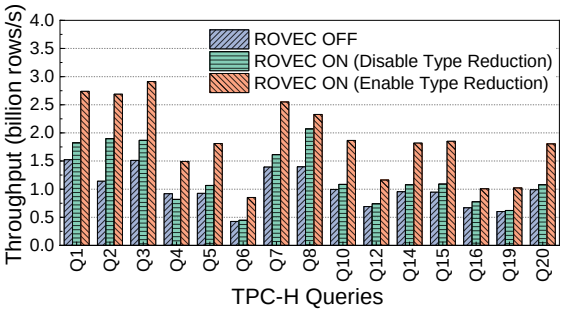


Fig. 7: Performance Break Down

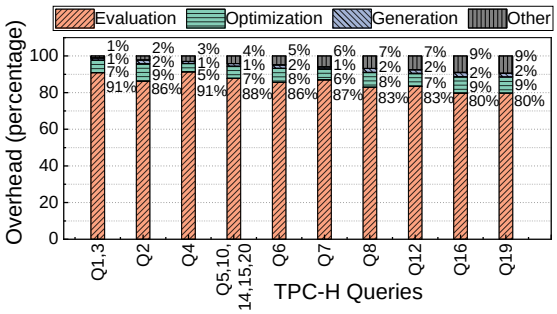


Fig. 8: Optimization Overhead

provided.

Storage and compression schemes. The data blocks with fixed number of tuples or fixed size in space are both feasible. By default, we inherit setting of PolarDB-C, which is the former setting, *i.e.*, fix number $(65,536)$ of tuples per block and also adopted by column databases like [4], [28]. The default chunk size is 4096 for each operator. Besides, FOR and GCD are used for data types of Integer, Decimal, and Date. Dictionary is used for string data, and the single-value scheme is used for all types whenever applicable.

5.2 Evaluation on TPC-H

We first evaluate all systems on TPC-H with extracted queries from Table 1 and 3 and summarize the results as follows: compared with PolarDB-C, ROVEC improves the throughput of table scan by up to 120% (60% on average) and reduces the latency of theta join by up to 50% (30% on average). Besides, ROVEC outperforms other systems significantly, *i.e.*, over $5\times$ average throughput improvement for table scan and theta join queries.

5.2.1 Table Scan

We first evaluate scan extracted queries from TPC-H as listed in Table 1. Then, we look into the detailed impact from each component of ROVEC, including the acceleration from type reduction and other optimizations as well as the overhead from block-wise optimization. The scalability and compression status (*e.g.*, compression ratio) are also studied. Finally, we study the impact of data skewness on ROVEC.

Overall table scan performance. Figure 6 shows the overall throughput of all systems on TPC-H. Overall, ROVEC achieves a significant improvement in throughput, which is about 180% of PolarDB-C on average.

First, we look at the case of single-thread evaluation as depicted in Figure 6(a). Compared with PolarDB-C, we observe that the throughput of ROVEC is increased by 82.5% on average. Among all issued queries, ROVEC achieves the largest improvement on $Q2$, because of the type reduction on column p_size from type $INTEGER$ to type $UINT8$. Compared with other comparing systems, the average throughput advantage of ROVEC grows even larger than 600% on average. This performance gap comes majorly from two aspects: (1) the runtime execution optimization from ROVEC; and (2) the system infrastructure of PolarDB-C, *i.e.*, the use of AVX-512. In particular, the impact of system infrastructure has been identified by comparing PolarDB-C with other systems, which is about 427% on average. Therefore, we conclude that the latest SIMD instructions bring in significant performance benefits, which can be further amplified by ROVEC.

Second, we turn to the evaluation with 32 threads. As shown in Figure 6(b), ROVEC improves the throughput by 40% on average compared with PolarDB-C and over 500% on average compared with other comparing systems. Compared one thread evaluation, we find that the performance gap between ROVEC and PolarDB-C tends to be smaller, which results from memory bandwidth bottleneck. Among all queries, ROVEC achieves the largest improvement on $Q10$ by 62%, mainly because of the type reduction on column $o\_orderdate$ from type `DATE` to type `UINT16`.

Break-down analysis of performance gain. Here we study the break-down performance improvements from different optimization components in ROVEC, such as type reduction, expression simplification, and unfolding. For type reduction, we build an intermediate version of ROVEC that

Index	Query
Q1	select count(*) from lineitem where l_shipdate $\leq$ date '1998-12-01' - interval '90' day
Q2	select count(*) from part where p_size = 30
Q3	select count(*) from orders where o_orderdate $\leq$ '1995-3-15'
Q4	select count(*) from lineitem where l_commitdate $<$ l_receiptdate
Q5	select count(*) from orders where o_orderdate $\geq$ date '1994-01-01' and o_orderdate $<$ '1994-01-01' + interval '1' year
Q6	select count(*) from lineitem where l_shipdate $\geq$ '1994-1-1' and l_shipdate $<$ '1994-1-1' + interval '1' year and l_discount between $.06-0.01$ and $.06+0.01$ and l_quantity <24
Q7	select count(*) from lineitem where l_shipdate between '1995-01-01' and '1996-12-31'
Q8	select count(*) from part where p_type = 'ECONOMY ANODIZED STEEL'
Q10	select count(*) from orders where o_orderdate $\geq$ date '1993-10-01' and o_orderdate $<$ '1993-10-01' + interval '3' month
Q12	select count(*) from lineitem where l_receiptdate $\geq$ date '1994-01-01' and l_receiptdate $<$ '1994-01-01' + interval '1' year and l_shipmode in ('MAIL', 'SHIP')
Q14	select count(*) from lineitem where l_shipdate $\geq$ date '1995-09-01' and l_shipdate $<$ '1995-09-01' + interval '1' month
Q15	select count(*) from lineitem where l_shipdate $\geq$ date '1996-01-01' and l_shipdate $<$ '1996-01-01' + interval '3' month
Q16	select count(*) from part where p_size in (49, 14, 23, 45, 19, 3, 36, 9)
Q19	select count(*) from lineitem where l_shipmode in ('AIR', 'AIR REG') and l_quantity $\geq$ 10 and l_quantity $\leq$ 10 + 10 and l_shipinstruct = 'DELIVER IN PERSON'
Q20	select count(*) from lineitem where l_shipdate $\geq$ date '1993-1-1' and l_shipdate $<$ date_add('1993-1-1', interval '1' year)

TABLE 1: Extracted TPC-H Table Scan Query

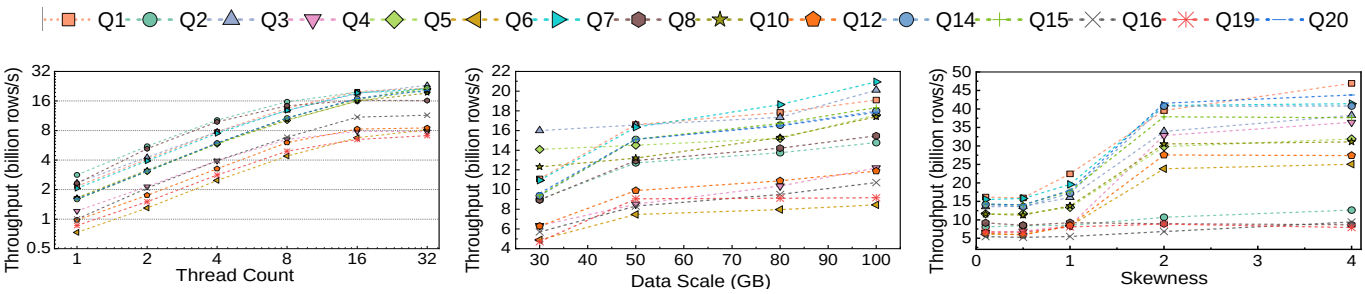


Fig. 9: Thread Scalability

Fig. 10: Data Size Scalability

Fig. 11: Throughput vs. Skewness

switches off the type reduction stage, and compare it with the full version of ROVEC in Figure 7. In the figure, we denote this intermediate version as ROVEC ON (Disable Type Reduction) and the full version as ROVEC ON (Enable Type Reduction). Moreover, to measure the performance gain from other optimization components (*e.g.*, expression simplification, and unfolding), we build a version without any optimization and denote it as ROVEC OFF.

Type reduction. As shown in Figure 7, type reduction is the dominant optimization contributing to the major performance gain for all queries. Specifically, ROVEC ON (Enable Type Reduction) increases the throughput by 57% compared with ROVEC OFF, and by 40% compared with ROVEC ON (Disable Reduction OFF). On average, over 80% of the performance gain is from this optimization. Especially, for query $Q19$, over 98% improvement is from type reduction.

Other optimizations. For all queries (except on $Q4$), we observe that other optimization components (*i.e.*, excluding type reduction) improve throughput by 17% on average and accounts for about 15% performance gain of ROVEC. In Figure 7, ROVEC (Disable Type Reduction) on $Q4$ performs worse than ROVEC OFF. This is because other optimizations hardly work when comparing two data fields so that the overhead of optimization outweighs the benefits. In summary, other optimizations also contribute to the throughput improvement but with limited impact.

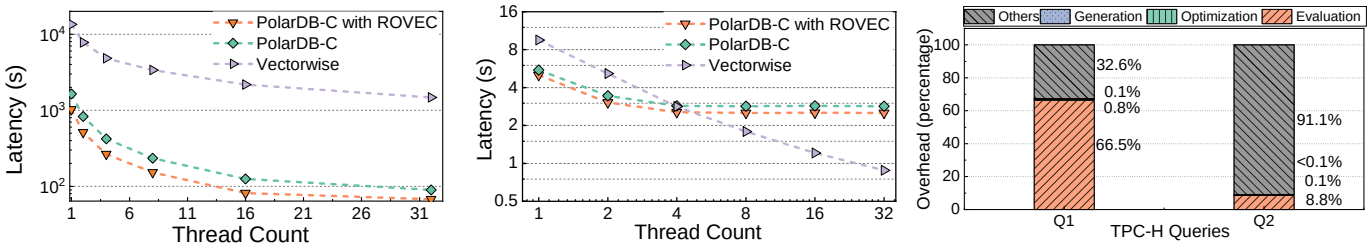
Optimization overhead. Here we study the optimization overhead of ROVEC, which is mainly from expression optimization for each input data block. Overall, the optimization overhead only accounts for less than 9% of the total running time, which is negligible compared with the performance gain it brought. To deeply investigate this overhead, we divide the entire query processing procedure into four stages: optimization (*i.e.*, Step $1-4$ in Figure 2), generation (*i.e.*, interpreting optimized expression into machine code as Step 5 in Figure 2 shows), evaluation (*i.e.*, data fetching and expression evaluation) and other miscellaneous processes (*e.g.*, query parsing and output handling). Figure 8 shows breakdown overheads of each stage in terms of the percentage of the entire query execution time. For conciseness, similar queries are combined as a group and it results in 10 groups in total. As shown in Figure 8, the optimization stage ranks second among all stages and only accounts for 7.45% of the total running time on average. The evaluation stage is the heaviest part, which accounts for 86% of the total running time. For the rest two stages, they only account for about 10%. In summary, the optimization overheads are small , and negligibly affects query processing.

Storage and Compression Schemes. The application of compression schemes can reduce the data volume transferred between memory and disk, which is at the cost of compression/decompression overhead. Therefore, the compression schemes should be employed if the reduced data loading time is larger than the compression and decompression time. Considering that our paper mainly targets an analytical database based on column store, the data volume transferred is huge but can be greatly reduced with compression schemes, whose (compression/decompression) overhead is negligible [11], [12], [13]. Thus, it's no wonder that lightweight compression schemes now have been the de-facto choice in column databases like Vectorwise, Hyper, and C-Store. Moreover, in our proposed optimization framework, the overhead is further reduced when integrated together with the expression evaluation process. In summary, in our framework, the compression schemes are applied by default.

In the evaluation, the applied encoding schemes depend on data types: all numeric columns are encoded with FOR; DATETIME with FOR and GCD; VARCHAR with Dictionary; and Single-value scheme is used whenever possible. To show the effectiveness of compression schemes, here we investigate the compression status (*e.g.*, compressed type and compression ratio) of data in TPC-H dataset. In general, most of the data can be compressed by element-addressable schemes, and all schemes in Section 2 are used in the evaluation. On average, 100GB raw data is compressed to 52GB files on disk, where the compression ratio is 2. To look into the compression status, we select frequently used columns

Column	SQL Type	Compression Scheme	Compression Ratio
l_linenumber	Integer	FOR, Single-value	4
l_quantity	Decimal(15,2)	FOR, Single-value	4
l_orderkey	Integer	FOR, Single-value	1.337
l_extended price	Decimal(15,2)	FOR, Single-value	2
l_discount	Decimal(15,2)	FOR, Single-value	8
l_receiptdate	Date	FOR, Single-value GCD	4.002
l_shipdate	Date	FOR, Single-value GCD	4.002
l_shipmode	Char(10)	FOR, Single-value Dictionary	10
l_shipinstruct	Char(25)	FOR, Single-value Dictionary	25

TABLE 2: Compression Status of Lineitem Table



(a) Performance on Q1 (b) Performance on Q2 (c) Optimization Overhead

Fig. 12: Overall Theta Join Performance on TPC-H

from table *Lineitem* and list their compression statuses in Table 2. There are $600,037,902$ rows in the table, and all columns are effectively compressed by ROVEC. For example, field $l\_linenumber$ is originally $INTEGER$ (SQL type) and compressed by FOR scheme, which makes all blocks stored as $UINT8$ (storage type). For field $l\_shipstruct$, it is compressed by Dictionary into $UINT8$ from $CHAR(25)$.

Scalability. Here we study the scalability of ROVEC, including both thread scalability and data size scalability. Note that few new contentions are introduced in by ROVEC since it only takes the tiny block summary and logical expression as input, whose accessing overhead is negligible (only memory access) compared with the evaluation overhead as shown in Figure 8. Therefore, the scalability of ROVEC generally goes with the underlying column stores.

Thread scalability. Figure 9 shows the thread scalability of ROVEC on 100G TPC-H dataset. Overall, the throughput increases as the thread count increases. Among all queries, $Q16$ gains the largest improvement (1292%), while $Q2$ gains the smallest (475%). As can be seen in Figure 9, the throughput increases by 894% on average when the thread count increases from 1 to 32. However, the throughput growth gradually slows down as the thread number increases. The system achieves its peak throughput when the number of threads reaches 16, and the throughput increment slows down and almost stagnates when the number of threads grows larger than 16. This is because the system has been saturated and the resource contention (especially the buffer pool contention we identified) from the underlying column store becomes severe.

Data size scalability. Figure 10 shows the data size scalability with 32 threads, and a 39% throughput improvement is observed when the data size increases from 30GB to 100GB. Overall, the throughput increases as the data size increases. This is because SIMD-based evaluation becomes beneficial when there are massive data blocks to be processed, which dominates the cost of the query execution.

Skewness. Here we study how the skewness of data distribution affects ROVEC. Since the default distribution of TPC-H is uniform, we generate a skewed TPC-H from the Zipf distribution tool [29] and set skewness factor to $0.1, 0.5, 1, 2$, and 4. The larger skewness factor indicates that the generated dataset concentrates on fewer distinct values. Figure 11 shows the throughput of ROVEC on all queries against different skewness factors. Overall, the throughput increases as the data skewness increases. Specifically, when the skewness factor increases from 0 to 4, the throughput improves by 353% on average. It is because that a skewed dataset leads to fewer distinct values in a data block, which makes compression schemes more effective. This situation further opens up more opportunities for the type reduction to convert values into smaller types, leading to higher throughput. The increased opportunity of compression then increases the possibilities of type reduction and thus the throughput of ROVEC. We observe that the throughput may suddenly encounter leap changes when the skewness reaches some certain degrees (*e.g.*, skewness factor from 1 to 2 in Figure 11). From a closer investigation, we find that such leaps are caused by more effective type reduction, *e.g.*, the type changes from $UINT32$ to $UINT16$.

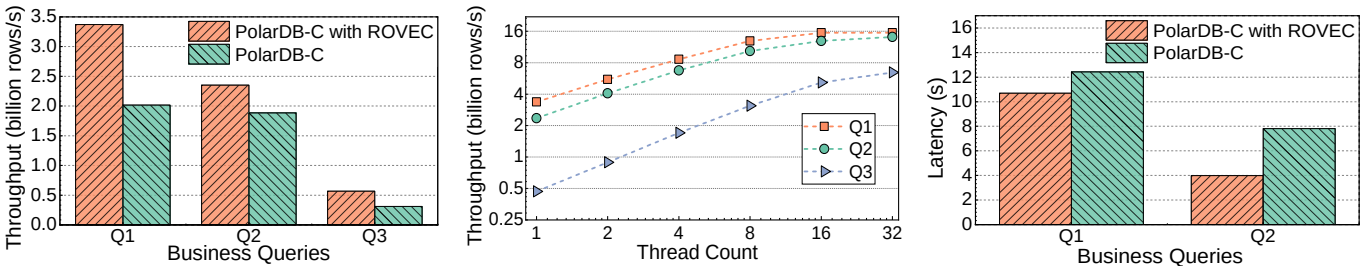
Index	Query
Q1	select count(*) from lineitem, orders where l_extendedprice > o_totalprice + 100000
Q2	select count(*) from customer, supplier where c_acctbal > s_acctbal and c_nationkey > s_nationkey

TABLE 3: Theta Join Queries for TPC-H

5.2.2 Theta Join

Considering that there is no theta join in TPC-H, we synthetically construct two theta join queries as listed in Table 3, which are inspired by [27]. As theta join is not supported by all compared systems (*e.g.*, Clickhouse and DB2-BLU), we present results of the representative system Vectorwise. Besides, since evaluations like scalability and skewness share similar trends to the results on the table scan operator, we omit them from this section.

Overall theta join performance. Overall, as shown in Figure 12(a) for $Q1$, ROVEC reduces the latency by 30% on average compared with PolarDB-C, and 90% latency compared with Vectorwise. For $Q2$ in Figure 12(b), the latency of ROVEC is 90% of PolarDB-C when the thread number increases from 1 to 32. Compared with Vectorwise, ROVEC reduces the latency by 50% for one thread but shows over $2\times$ latency with 32 threads. With closer observation, we find that the performance stagnating of PolarDB-C with multithread is due to resource contentions (especially the buffer pool of the underlying column store) while Vectorwise shows better scalability. Similar to that of the table scan operator, the performance gain also mainly comes from the full exploitation of SIMD and runtime optimizations from ROVEC. Especially, SIMD instructions (*i.e.*, AVX-512) greatly accelerate the execution by one order of magnitude, and the performance gain is further amplified by ROVEC (*i.e.*, type reduction) over 30%.



(a) Table Scan Performance (b) Scan Scalability Performance (c) Theta Join Performance

Fig. 13: Evaluations on Real-world Dataset RO

Index	Query
Q1	select count(*) from itm_detail where star_id = 5
Q2	select count(*) from itm_detail where main_cnt≤990 and cr_rank=3
Q3	select count(*) from itm_detail where level1 in (50002766, 50016422,50050359, 124458005, 50026316, 500081841) and time_len_001 ≥ 10 and time_len_001 ≤ 15) and (cnt_030 between 300 and 999)

TABLE 4: Table Scan Queries for RO

Index	Query
Q1	select count(*) from cust_dbm a, app_dbm b where a.pv7 < b.pv7 and a.pv30 < b.pv30
Q2	select count(*) from cust_dbm a, app_dbm b where a.cnt7 between b.cnt14 and b.cnt30 and a.cnt7 > b.cnt7

TABLE 5: Theta Join Queries for RO

Optimization overhead. As shown in Figure 12(c), the optimization overhead accounts for about 0.8% of the total execution time on Q1 and about 0.1% for $Q2$. The reason behind such low overhead is the explosion of evaluation workload compared with the table scan. In particular, different from the single loop in the table scan, theta join requires a nested loop through two columns, which takes much more time and thus significantly outweighs the percentage of optimization overhead. Moreover, other costs in $Q2$ accounts for more than 91.1% of the running time, which is caused by the management of large output generated by theta join. In summary, the optimization overhead ($< 1\%$) is negligible compared with the evaluation cost ($> 30\%$).

5.3 Evaluation on Real-world Dataset

In our evaluation, RO is compressed from 192GB to 100GB and Table itm_detail involved in table scan queries contains over 150M rows. For the two tables involved in queries in Table 4, there are $335,073$ and $39,407$ rows in table app_dbm and $cust\_dbm$, respectively. The evaluation results are summarized as follows: compared with original PolarDB-C, ROVEC improves the throughput of table scan by 58% and reduces the latency of theta join by 30% on average.

Here, on RO dataset, only the evaluation results of PolarDB-C and ROVEC are presented for the following two reasons. Firstly, RO is collected from the production environment of an online recommendation system and is disallowed to be imported into systems unavailable in the production environment. Secondly, as ROVEC is implemented on top of a column database (PolarDB-C as an example), the provided evaluation results of ROVEC and PolarDB-C are already sufficient to show the performance gain brought by ROVEC.

5.3.1 Table Scan

Overall performance. We evaluate ROVEC on RO with queries in Table 4. As shown in Figure 13(a), ROVEC improves the throughput by 58% on average compared with PolarDB-C. For $Q1$ only containing an equality predicate and $Q2$ involving conjunction, the throughput is increased by 67% and 24%, respectively. As for the most complex query $Q3$, the throughput is increased by over 80%.

Thread scalability. As shown in Figure 13(b), when the thread count increase from 1 to 32, the throughput of ROVEC improves by 700% on average, *i.e.*, 360% on $Q1$, 500% on $Q2$, and 1126% on $Q3$. Besides, similar to that on TPC-H, the throughput growth slows down as the threads increase and even stalls when reaching 16 threads, which is due to the system saturation and the resource contention among threads. Specifically, when the thread count increases from 1 to 32, the throughput improves by 700% on average, *i.e.*, 360% on $Q1$, 500% on $Q2$, and 1126% on $Q3$ respectively.

5.3.2 Theta Join

Overall, ROVEC shows significant improvement for theta join, *i.e.*, reducing the latency by 30%. Specifically, $Q1$ and $Q2$ are the two commonly used queries in the recommendation system. As shown in Figure 13(c), ROVEC reduces the latency by 15% for $Q1$ and by 50% for $Q2$ compared with PolarDB-C. As shown in Figure 7, the performance gain comes mostly from reducing type, which means the performance gain is limited by the degree of data compression. The performance gain of $Q1$ is limited by 15% as the field $pv7$ and $pv30$ is only partially compressed, which leads to limited performance gain. Meanwhile, as the field $cnt7$ from table $cust\_dbm$ and $cnt7, cnt14, cnt30$ from app_dbm are well compressed, the performance gain is increased by 50%.

6 Related Work

Expression evaluation optimization. Expression evaluation, included in operations like scans, joins, and predicates, is often the dominant cost of query execution [30]. We also note that Bitweaving is extraordinary for fast scan [8]. However, the functionality coverage of Bitweaving is limited, as it only handles boolean predicates and loses efficacy when predicates contain arithmetic operators, *e.g.*, $a + b > 100$. Besides, it has been shown that Bitweaving may suffer from

point accesses and scans with low selectivity [31]. In contrast, ROVEC aims at dynamically reducing the evaluation type of intermediate data during evaluation. We also find that a specific type optimization also reveals in DB2-BLU [9]. However, DB2-BLU is significantly different from ROVEC in (1) DB2-BLU relies on a customized compression scheme and storage format while ROVEC works with common and simple lightweight compression schemes and storage format; (2) more importantly, similar to Bitweaving, the function coverage of DB2-BLU is limited to boolean (or leaf) predicates. Note that SIMD-oriented type optimization is also discussed in Vectorwise [10]. However, with careful comparison, we find the type optimization in Vectorwise is totally different from ROVEC. Vectorwise chooses the smallest type to represent the fields in queries according to the value domain after direct scanning and decompression [10]. Firstly, such type optimization may incur heavy overheads from scanning and decompression while ROVEC only relies on the lightweight block-wise statistics in block summaries without scanning and decompression. More importantly, Vectorwise ignores the opportunities that the intermediate data types during evaluation could be could be cast to smaller (smaller than the input data) types and benefits later evaluations. For example, given expression $a(int32) - b(int32) > c(int16)$, if the result of $a - b$ is covered by type $int16$, then a purposely added cast $a - b$ to $int16$ would be more beneficial to SIMD-based evaluation than cast c to $int32$. Finally, the situation may get worse for Vectorwise if no smaller types can be found as it burdens the overheads but obtains no benefits. In contrast, instead of decompression, ROVEC dives into the decompression layer and brings the type reduction into the integration of decompression and expression evaluation.

Compressed databases. Accessing data from disks incurs heavy I/O cost and has been a heavy burden to many database systems. Compression is a canonical solution to reduce the data volume on disk and transferred to memory and has been widely adopted by databases for I/O-intensive workloads [32], [33], [34], [35]. In addition, compression also saves storage space on disk. There have been many works [32], [33], [34], [35] studying the impact of compression techniques over database systems. Willhalm *et al.* [7] proposed a SIMD-friendly scheme to decompress and scan compressed data, which is however limited to the bit-packed compression. Another work [36] is also a variant of the bit-packed compression scheme. These existing approaches focus on the optimization of the decompression process, which is independent of the subsequent expression evaluation; and other techniques either rely on extra storage space or a customized storage format. On the whole, these algorithms focus on the optimization of the decompression process and fail to optimize the decompression and evaluation as a whole. In contrast, ROVEC aims at the co-optimization of decompression and evaluation processes, which utilizes type reduction from lightweight compression schemes to speed up SIMD-based evaluation.

Queries on compressed data. As for column database, the data is stored column by column and thus increases the similarity of adjacent data together with probability of getting compressed. Benefited by the storage layout of the column database, lightweight compression schemes [13] can achieve a high compression ratio with low CPU cost [5]. Hence, they are far more popular in column stores compared with heavy compression algorithms like GZip [20] and LZO [37]. Previous works [5], [38] show how to query on compressed data directly, but only dictionary compression is supported. In contrast, ROVEC not only supports queries on compressed data with dictionary schemes, but also other schemes like FOR, GCD and *etc.*. SBoost [5] embeds multiple compression schemes (including run-length, bit-packed, dictionary, and delta), which is a storage engine supporting data filtering on compressed data. However, it is a storage engine with the ability of filtering data while ROVEC is a runtime optimization framework in the execution layer supports multiple operators like table scan and join. Apart from that, the predicate supported by SBoost only includes comparison between compressed field and a constant, which falls into a corner case of predicates supported by ROVEC. However, ROVEC seeks to reduce the evaluation type during general expressions evaluation. Lang *et al.* proposed to build a customized index that narrows the scan range [31], which however introduces extra IO and lookup cost when the narrowed range is small. Besides, the supported query types are rather limited (*e.g.*, predicates between two data fields are not supported). Ghita *et al.* proposed a white-box compression model that exposes compression status to query executor [39]. However, it only enables limited filter predicate push-down, while ROVEC aims at the optimization of general expression evaluation.

7 Conclusion

In this paper, we study the optimizations of expression evaluation in column databases and propose a runtime optimization framework named ROVEC. The key insight is to delay the expression optimization to execution time and deeply exploit fine-grained block summary to optimize expression evaluation. ROVEC aims at reducing evaluation data type at the unit of a data block, which makes SIMD-based evaluations operate on more values in each instruction. We extend ROVEC to support various operators and different data types, to offer general-purpose optimization capacity for real-world applications. To validate the effectiveness of ROVEC, we integrate ROVEC into a column database PolarDB-C. The evaluation results show that ROVEC achieves up to 125% (60% on average) throughput improvement for table scan and up to 50% (30% on average) latency improvement for theta join.

Acknowledgments

This work was done and completed at Alibaba. This work was supported in part by China Scholarship Council under Grant 202006190216, in part by the National Natural Science Foundation of China under Grant 61872178, in part by the Natural Science Foundation of Jiangsu Province under Grant No. BK20181251, in part by the open research fund of Key Lab of Broadband Wireless Communication and Sensor Network Technology (Nanjing University of Posts and Telecommunications), in part by the Key Research and Development Project of Jiangsu Province under Grant No. BE2015154 and BE2016120, in part by the National Natural Science Foundation of China under Grant 61832005.

REFERENCES

[1] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *SIGMOD*. ACM, 2015, pp. 1917–1923.

[2] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang *et al.*, "The snowflake elastic data warehouse," in *SIGMOD*. ACM, 2016, pp. 215–226.

[3] M. Zukowski, M. van de Wiel, and P. Boncz, "Vectorwise: A vectorized analytical dbms," in *ICDE*. IEEE, 2012, pp. 1349–1350.

[4] D. Ślezak, J. Wróblewski, V. Eastwood, and P. Synak, "Brighthouse: an analytic data warehouse for ad-hoc queries," in *VLDB*. VLDB Endowment, 2008, pp. 1337–1345.

[5] H. Jiang and A. J. Elmore, "Boosting data filtering on columnar encoding with SIMD," in *DaMoN*. ACM, 2018, pp. 6:1–6:10.

[6] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates." in *ADMS*. VLDB Endowment, 2013, pp. 1–12.

[7] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units," in *VLDB*. VLDB Endowment, 2009, pp. 385–394.

[8] Y. Li and J. M. Patel, "Bitweaving: fast scans for main memory data processing," in *Proceedings of the International Conference on Management of Data*. ACM, 2013, pp. 289–300.

[9] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman *et al.*, "Db2 with blu acceleration: So much more than just a column store." VLDB Endowment, 2013, pp. 1080–1091.

[10] J. Sompolski, M. Zukowski, and P. Boncz, "Vectorization vs. compilation in query execution," in *DaMoN*. ACM, 2011, pp. 33–40.

[11] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: An experimental survey (experiments and analyses)." in *EDBT*. Springer-Verlag, 2017, pp. 72–83.

[12] P. Damme, A. Ungethüm, J. Hildebrandt, D. Habich, and W. Lehner, "From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms," *ACM, Transactions on Database Systems*, vol. 44, no. 3, p. 9, 2019.

[13] J. Hildebrandt, D. Habich, T. Kühn, P. Damme, and W. Lehner, "Metamodeling lightweight data compression algorithms and its application scenarios," in *Proceedings of the ER Forum and the ER Demo Track co-located with the International Conference on Conceptual Modelling*. Springer, 2017, pp. 128–141.

[14] C. Lattner, "Llvm," www.aosabook.org/en/llvm.html.

[15] I. Free Software Foundation, "Gcc," gcc.gnu.org/.

[16] Wikipedia, "Theta join," en.wikipedia.org/wiki/Relational_algebra.

[17] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, Z. Liu, and T. Zhang, "PolarDB meets computational storage: Efficiently support analyticalworkloads in cloud-native relational database," in *FAST*. USENIX, 2020, pp. 29–41.

[18] W. Cao, Z. Liu, P. Wang, S. Chen, C. Zhu, S. Zheng, Y. Wang, and G. Ma, "PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database," *VLDB*, pp. 1849–1862, 2018.

[19] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes," in *ICDE*. IEEE, 1998, pp. 370–379.

[20] M. A. Jean-loup Gailly, "Gzip," www.gzip.org.

[21] TPC-H, "Tpc," tpc.org/tpch.

[22] Actian, "Vectorwise community edition," https://www.actian.com/lp/vector-community-edition/.

[23] YANDEX, "Clickhouse," clickhouse.yandex.

[24] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh, "Quickstep: A data platform based on the scaling-up approach," *VLDB*, p. 663–676, 2018.

[25] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, "Hyrise: A main memory hybrid storage engine," *VLDB*, p. 105–116, 2010.

[26] M. Zukowski and P. A. Boncz, "Vectorwise: Beyond column stores," *IEEE Data Engineering Bulletin*, pp. 21–27, 2012.

[27] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *SIGMOD*. ACM, 2011, pp. 949–960.

[28] J. Johnson and G. Johnson, "Building knowledge around complex objects using inforbright data warehousing technology," *International Journal of Database Theory and Application*, pp. 31–46, 2010.

[29] F. Yu, "Skewd tpc-h dataset generator," github.com/YSU-Data-Lab/TPC-H-Skew.

[30] P. Boncz, T. Neumann, and O. Erling, "Tpc-h analyzed: Hidden messages and lessons learned from an influential benchmark," in *TPCTC*. Springer, 2013, pp. 61–76.

[31] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper, "Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation," in *SIGMOD*. ACM, 2016, pp. 311–326.

[32] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems," in *SIGMOD*. ACM, 2001, pp. 271–282.

[33] B. R. Iyer and D. Wilhite, "Data compression support in databases," in *VLDB*. VLDB Endowment, 1994, pp. 695–704.

[34] G. Ray, J. R. Haritsa, and S. Seshadri, "Database compression: A performance enhancement tool," in *COMAD*. Tata McGraw-Hill, 1995.

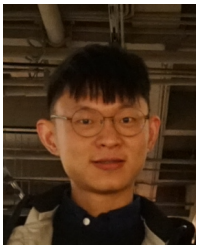
[35] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases," *ACM, Sigmod Record*, pp. 55–67, 2000.

[36] D. Lemire and L. Boytsov, "Decoding billions of integers per second through vectorization," *Wiley Online Library, Software: Practice and Experience*, pp. 1–29, 2015.

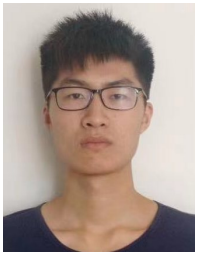
[37] Oberhumer, "Lzo," www.oberhumer.com/opensource/lzo/.

[38] D. J. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*. ACM, 2006, pp. 671–682.

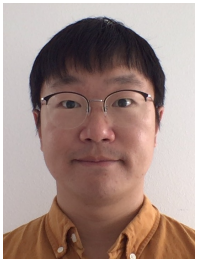
[39] B. Ghita, D. G. Tomé, and P. A. Boncz, "White-box compression: Learning and exploiting compact table representations." in *CIDR*, 2020.



Meng Li received his B.S. degree in Computer Science from Nanjing University, Jiangsu, China, in 2016. He is currently a Ph.D. student at Nanjing University. His research interests are in the area of high-performance query processing in databases.



Zheyu Miao received B.S. degree in electrical engineering from Xi'an Jiaotong University, Xi'an, China, in 2016. He is currently a Ph.D. student at Zhejiang University, Hangzhou, China. His research interests are in the area of storage engines on decoupled storage and computation architecture.

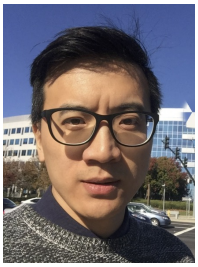


Di Wu received his B.S in Microelectronics from Xidian University, in 2009, and M.S degree in electrical engineering from Seoul National University, Seoul Korea in 2011. He has published three papers on customized SoC instruction design topics between 2009 and 2011, and one paper received ISOCC 2010 best paper award. After graduation in 2011, he focused on developing the query processing modules and has been a core member of SAP HANA, AliCloud PolarDB, and AWS Aurora.

Feifei Li received the BS degree in computer engineering from Nanyang Technological University, in 2002, and the PhD degree in computer science from Boston University, in 2007. He is currently a Vice President of Alibaba Group, ACM Distinguished Scientist, President of the Database Products Business Unit of Alibaba Cloud Intelligence, and Director of the Database and Storage Lab of DAMO academy.

Sheng Wang obtained Ph.D. in Computer Science from National University of Singapore. Currently, he is a Research Scientist in the Database and Storage Lab at Alibaba DAMO Academy. His research interests generally lie in building reliable, secure, intelligent, performant, and globally distributed database systems and services.

Wei Cao received the BEng and MEng degrees from Peking University, in 2006 and 2009, respectively. He is currently a senior technical expert and the technique leader of Aliyun RDS with Alibaba Group, Hangzhou, China. His research interests include databases, distributed systems, and cloud computing.



Zhi Qiao received the MS degree in Telecommunication from XiDian University in 2013. Currently, he is a senior engineer at the Database Department of Alibaba Cloud, China. His research interests include SQL pre-compiler, columnar vectorization execution, and Hybrid Transaction and Analytical Process(HTAP) in the database management systems.

Yukun Liang received BS degree from Zhejiang University in 2019. Currently, he is a PolarDB kernel developer of Alibaba. He is broadly interested in executor and expression evaluation in columnar databases.

Yubin Ruan received his B.S. degree in software engineering from South China University of Technology, Guangzhou in 2018. His interest lies in the database management systems. After graduation in 2018, he has been focusing on developing query processing modules in AliCloud PolarDB.

Jimmy Yang is currently a team lead for PolarDB Storage Engine in Alibaba Cloud. Before joining Alibaba, he was an InnoDB Architect in Oracle. He also worked in Sybase's Enterprise Server team as the index team manager.

Haipeng Dai received the B.S. degree in the Department of Electronic Engineering from Shanghai Jiao Tong University, Shanghai, China, in 2010, and the Ph.D. degree in the Department of Computer Science and Technology in Nanjing University, Nanjing, China, in 2014. His research interests are mainly in the areas of wireless charging, mobile computing, and data mining. He is an associate professor in the Department of Computer Science and Technology at Nanjing University. His research papers have been published in many prestigious conferences and journals such as ACM MobiSys, ACM MobiHoc, ACM VLDB, IEEE ICDE, ACM SIGMETRICS, ACM UbiComp, IEEE INFOCOM, IEEE ICDCS, IEEE ICNP, IEEE SECON, IEEE IPSN, IEEE JSAC, IEEE/ACM TON, IEEE TMC, IEEE TPDS, and IEEE TOSN. He is an IEEE and ACM member. He serves/ed as Poster Chair of the IEEE ICNP'14, Track Chair of the ICCCN'19, TPC member of the ACM MobiHoc'20-21, IEEE INFOCOM'20-21, IEEE ICDCS'20-21, IEEE ICNP'14, IEEE IWQoS'19-21, IEEE IPDPS'20 and IEEE MASS'18-19. He received Best Paper Award from IEEE ICNP'15, Best Paper Award Runner-up from IEEE SECON'18, and Best Paper Award Candidate from IEEE INFOCOM'17.

Guihai Chen received a B.S. degree in computer software from Nanjing University in 1984, an M.E. degree in computer applications from Southeast University in 1987, and a Ph.D. degree in computer science from the University of Hong Kong in 1997. He is a professor and deputy chair of the Department of Computer Science, Nanjing University, China. He had been invited as a visiting professor by many foreign universities, including Kyushu Institute of Technology, Japan in 1998, University of Queensland, Australia in 2000, and Wayne State University, USA from Sept. 2001 to Aug. 2003. He has a wide range of research interests focusing on sensor networks, peer-to-peer computing, high-performance computer architecture, and combinatorics.