Vectorization Support in Flink

1. Introduction

Vectorization refers to techniques for storing, manipulating and processing data in vectorized format. To make it simple, suppose we have a dataset of *n* records, and each record has *m* fields. Vectorization represents the dataset as *m* vectors. Each vector corresponds to one field of the dataset, and each vector has length *n*.

In contrast, row-store adopts a dual representation for the dataset. In particular, the dataset is represented by *n* tuples, each corresponding to a record. Each tuple consists of *m* elements, corresponding to the *m* fields of the record.

In recent years, vectorization is gaining more and more popularity in SQL communities. Systems like IBM DB2 [1], Microsoft SQL Server [2], and VectorWise [3] adopts this model. However, Flink is adopting row-store in the SQL operators, as well as the task framework.

2. Benefits of Vectorization

Compared with row-store, vectorization has the following benefits:

1. Vectorization leads to better CPU efficiency. Vectorization processes all elements in a single vector in sequence. This has the following consequences:
   a) This makes better use of modern CPU's data prefetching feature, and leads to better data locality and more cache hits.
   b) This processing style is also friendly to i-cache.
   c) In addition, vectorization makes is much easier to use SIMD [4], which will improve performance significantly. Note that Java starts to support SIMD in Java 8, and it is expected that high versions of Java will provide better support for SIMD [5].

2. Vectorization leads to better IO efficiency. For most SQL queries, only a few fields are used in the query processing. However, for row-store, all data must be loaded from external store. In contrast, for vectorization, only the related vectors need to be loaded, significantly reducing the IO overhead [6].

3. Vectorization provides more compact memory layout. For fixed-length data types, there is almost no extra memory overhead. However, this is unlikely to happen with row-store, because even if only one field has variable length, the whole record has variable length. Compact memory layout leads to two results:

   a) With the same amount of memory space, more data can be loaded into memory, and fewer spills are required. This leads to better performance.

b) The cost of shuffling can be reduced significantly. This leads to better performance and less network IO.

4. Vectorization makes it much easier for data compression [7], which will reduce the memory requirements and shuffle costs. To see this, note that popular data formats for big data, like Parquet and Orc are both based on vectorization.

3. A Hybrid Architecture

Despite the benefits of vectorization described above, there are some scenarios for which row-based approach performs better (e.g. some hash join operators), so a purely vectorized architecture may lead to performance degradations for such scenarios. To overcome this problem, our proposal is primarily based on vectorization, but also supports row-store, through the following methods:

1) Provide a row-based view on column data.
2) Provide adapters to transform vector data to row data, and vice versa.
3) If the above two methods are not efficient enough, the optimizer may decide to process the task completely in row-mode, with row/vector transform carried out by serialization/deserialization at task boundaries.

These methods can be chosen statically or dynamically. Note that method 1 is already provided in Flink (by ColumnarRow and VectorizedColumnBatch classes), and in our initial implementation (described in the next section), method 2 is also provided. Comparisons of these methods are summarized in the table below:

Table 1

Solution	Pros.	Cons.
Row-based view	● No need to copy underlying data. So the cost for copying data is saved. ● No extra memory required.	The underlying data is still columnar, so there can be cache misses when accessing multiple fields of a record.
Row/Vector adapter	● Involving performance overhead for data copying during row/vector transform. ● Extra memory is required.	The underlying data is row-based, so there is good data locality for accessing fields of a row.
Row-mode task	● No extra data copy or data transform. Data format transform is carried out by serialization/deserialization.	The cost model may not be accurate, which may lead to sub-optimal solution.

Vectorized data can be processed in two ways: *batch-based processing* vs. *pure-vectorized processing*. Let us illustrate with an example. Suppose we have the following pseudo-code for row-based processing:

```
    public void processElement(StreamRecord record) {
        BaseRow row = (BaseRow) record.getValue();

        Process field 1
        Process field 2
        …
        Process field m
    }
```

Batch-based processing is simply wrapping row-oriented processing in a loop:

```
    public void processBatch(StreamRecord record) {
        RecordBatch batch = (RecordBatch) record.getValue();
        for (int i = 0; i < batch.getSize(); i++) {
                BaseRow row = (BaseRow) batch.getRow(i);

                Process field 1
                Process field 2
                …
                Process field m
        }
    }
```

In contrast, pure-vectorized processing deals with each field separately:

```
    public void processBatch(StreamRecord record) {
        RecordBatch batch = (RecordBatch) record.getValue();

        Vector vec = batch.getVector(1);
        for (int i = 0; i < batch.getSize(); i++) {
                Process field 1
        }
        Vector vec = batch.getVector(2);
        for (int i = 0; i < batch.getSize(); i++) {
                Process field 2
        }

        …

        Vector vec = batch.getVector(n);
        for (int i = 0; i < batch.getSize(); i++) {
                Process field n
        }
   }
```

For some scenarios, pure-vectorized processing has performance advantages over batch-based processing (e.g. easier to apply SIMD, see [5]). However, this proposal is mainly focusing on batch-based processing, and we have a longer way to go before we completely support pure-vectorized processing in Flink.

It should also be noted that, even with batch-based processing, we can achieve notable performance gain compared with row-based processing. As an example, note that for row-based processing, each row goes through an operator chain by invoking a series of virtual functions. So there are a number of operations for pushing/popping the call stack, virtual function resolution, etc., just for a single row. By processing records in a batch, the performance overhead of these operations can be amortized.

Detailed performance results of our initial implementation (which is mainly based on batch-based processing) will be given in the next section.

4. Initial Results with Vectorization

We have developed an initial implementation for vectorization, based on a historical version of Blink (Many thanks to @Kurt Young’s team for all the kind help). We have used both methods described in Section 3 to support row-based processing, and our initial implementation is primarily batch-based processing, as described in the previous section.

We evaluate the performance of TPC-H queries (1TB), on both row-store and vectorization. The experiments were performed in a cluster with 21 nodes. Each node had 64 cores, 256 GB memory and 10 Gb/s network card. The results are as given in Fig. 1, in second (Please note that at the time of this writing, not all TPC-H queries are supported yet). It can be observed that vectorization leads to an average performance improvement of 30%. Therefore, it can be seen that notable benefits can be expected by supporting vectorization in Flink.

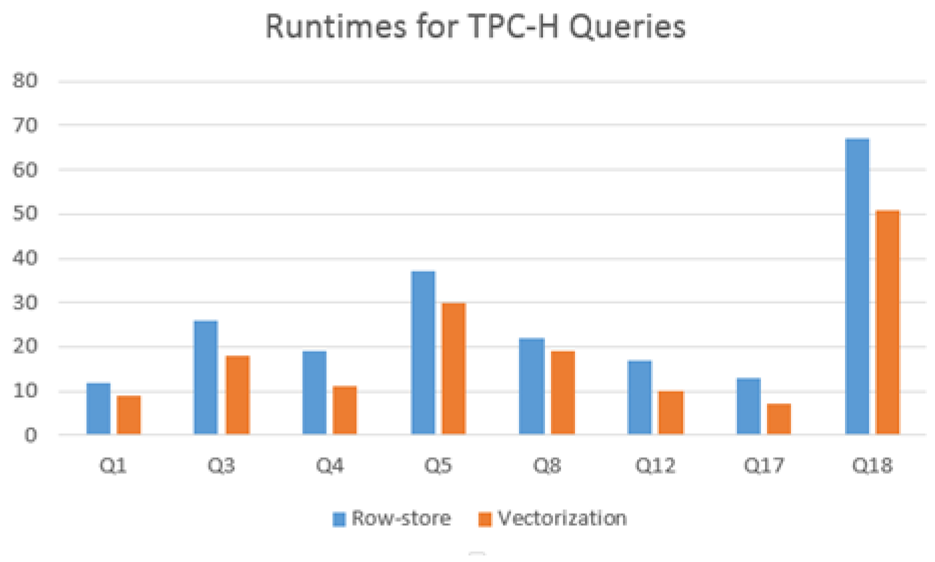

Fig. 1

5. Implementation Concerns

The biggest challenge for introducing vectorization to Flink is the substantial changes needed for the existing code base. To address this, we split the changes into a few small incremental changes, which makes the whole process easy to manage. Before describing the incremental changes, let’s have a detailed review of the current status.

5.1 Current Status

Fig. 2 shows the basic structure of a typical Flink task. It contains an operator chain with multiple operators. If the task has no upstream task, the task input is obtained from a data source. Otherwise, the input is from an input gate. Similarly, the task output is sent to a data sink, if it has no downstream task. Otherwise, the output is sent to downstream tasks through a record writer. Intermediate data produced in each step are in row format. Let’s take a closer look into each step.

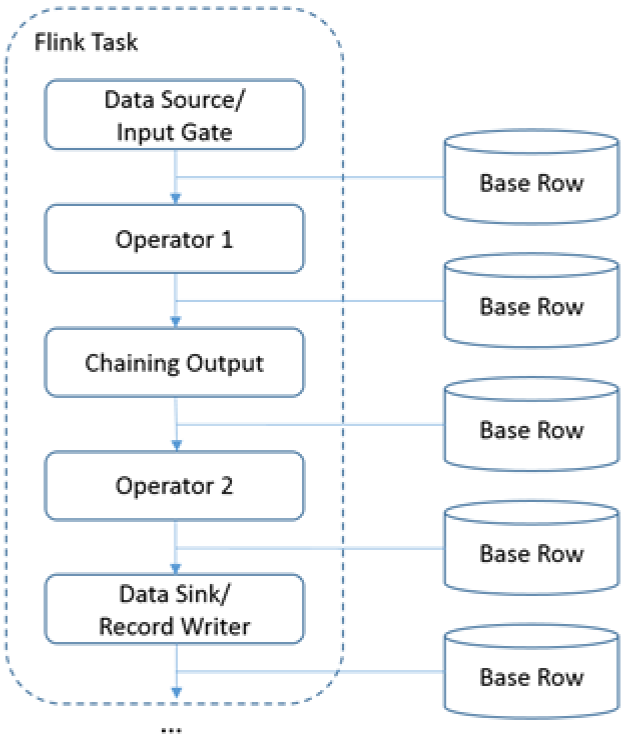

Fig. 2

5.1.1 Reading Input Data

The task reads input data from a data source or an input gate, depending on if it has upstream tasks. The input data is parsed as a stream of *base rows* (corresponding to interface org.apache.flink.table.dataformat.BaseRow), which are represented as row-store. Each base row is sent to the operator by calling the *processElement* method of the operator.

Notes:

1. For some input format (like Parquet and Orc), the input data are read as columnar row (corresponding to class org.apache.flink.table.dataformat.ColumnarRow). For ColumnarRow, the underlying memory layout is vectorized.
2. It appears that the based row comes out one by one. However, there are some underlying buffers in the network/file system layer, so essentially the input data are read in batch.

5.1.2 Processing Data by Operators

Once an operator receives a row data, it process it according to its own logic. In the process, one or more new rows can be generated, which is passed to the *collect* method of the *output* field:

```
output.collect(outRow);
```

If this operator is not the last in the operator chain, the type of the output field is ChainingOutput, which is an inner class defined in org.apache.flink.streaming.runtime.tasks. OperatorChain. The logic for ChainingOutput#collect method is simply passing the generated row to the *processElement* method of the next operator.

If the operator is the last one in the operator chain, the new row will be pushed to the data sink/record writer. We discuss this next.

5.1.3 Writing Output Data

When the last operator in the chain generates a row, it reaches the data sink to generate the job output, or it reaches the record writer to pass the data to downstream tasks.

Either way, the row is buffered and written in batch (in the network/file system layer), although it appears that the row is written one by one.

5.2 Our Design of the Incremental Changes

We support vectorization in an incremental manner. That is, each step involves a small amount of changes. Before giving details of the steps, please note that we try to follow the principles below:

- We primarily focus on batch jobs, although the changes can be extended to stream jobs in the future.
- We must provide a flag, which can be used to turn on/off the features in the steps, so Flink users can easily turn them off if they like to.
- The batch size/vector length must be configurable, so when the batch size is 1, vectorization degenerates to row-store.

5.2.1 Step 1 - Making Data Source/Shuffle Reader Generate Batch Data

With this step, the data generated by data source/input gate will be transformed to columnar layout. However, the columnar data provides a row-based view so that it does not require changes to operator interface.

Note that for some data format (like Parquet and Orc), the input is already in columnar format, so no change is required here. For other data format (like CSV), we need to attach a buffer to the data source/input gate. The details are illustrated in the following figure, with the changes highlighted.

Implementation Concerns:

1. The columnar layout can be based on org.apache.flink.table.dataformat.vector. VectorizedColumnBatch, and the row-based view can be org.apache.flink.table.dataformat. ColumnarRow. Both are already available in Flink.
2. A buffer can be attached to the data source/input gate, or an operator can be inserted to the head of the operator chain, which buffers the received rows, and transform them to batches.
3. In the process of adding row data to the buffer, the row may be deserialized to binary format (this depends on the row type, GenericRow needs deserialization while BinaryRow does not), and if the

data type is use-defined, memory footprint of records can increase significantly which in turn may cause OutOfMemoryException. This can be solved by setting the max batch size to 1 (Many thanks to @ Piotr Nowojski for the kind reminder).

Performance Impacts:

Buffering row data may cause delays in receiving the input data. However, the impact is insignificant, because before this change, there were also buffers in the underlying network/file system layers. This change just moves the buffers to Flink.

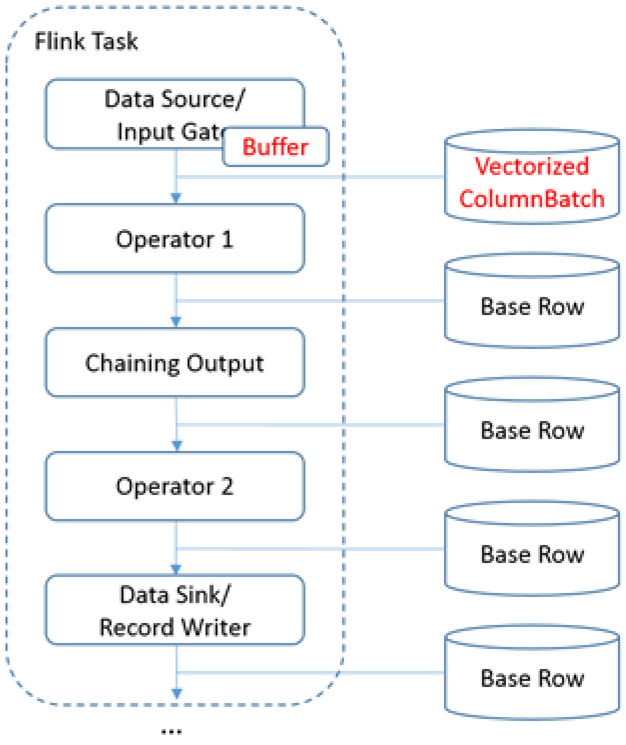

Fig. 3

5.2.2 Step 2 – Making Data Sink/Record Writer Accept Batch Data

In this step, the input of data sink/record writer changes from row to batch. So after this step, row format is only used internally in an operator chain. To transform row data to batch data, a buffer is needed at the data sink/record writer. Please note that the buffer in input gate is no longer required, since the upstream data are written as batches.

Implementation Concerns:

1. To accept batch data, there is no need to change interfaces for Data Sink/Record Writer, since these classes use template as the record type.

2. Serialization/Deserialization for batch data should be implemented, and such implementation should replace the logic for serializing/deserializing row data.
3. A buffer should be added to the data sink/record writer, or an operator should be placed at the end of the operator chain to transform row data to batch data.
4. When the processing of a task is finished, we must design a mechanism to notify the data sink / record writer about it. So the data sink / record writer will immediately send whatever it currently has to the output /downstream. There are two ways to do this:
    a) Explicitly call the flush method of the record writer, or the close method of the data sink.
    b) By sending a special record to indicate the end of processing.

Performance impact:

Buffering row data may cause delay for writing output data. However, the impact is insignificant, because before this change, there were also buffers in the underlying network/file system layer. This change just moves the buffers to Flink.

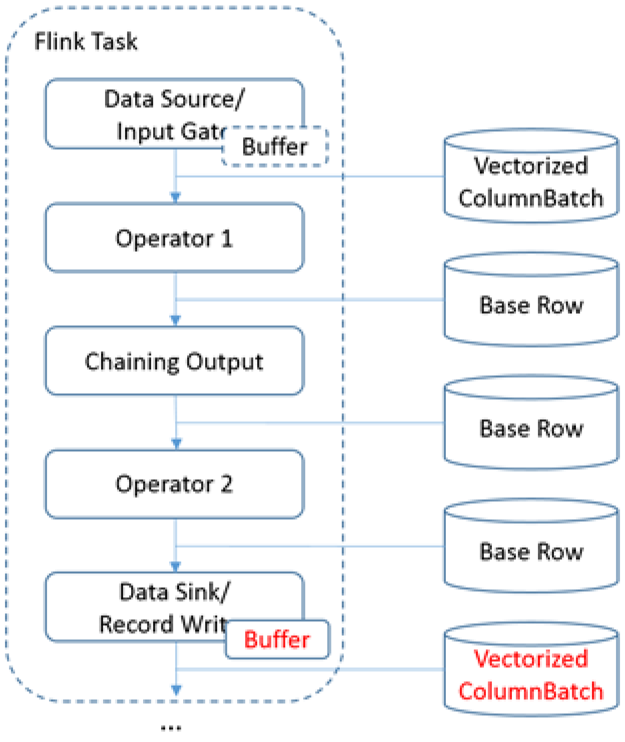

Fig. 4

5.2.3 Step 3 – Replacing Rows with Batches between Operators

In this step, the ChaningOuptut will be responsible for transforming output rows from the previous operator to batches as input to the next operator. The batches have a row-based view, so there is no need to change the operator interface.

Implementation Concerns:

1. Attaching buffers to ChainingOutput can be accomplished by sub-classing this class with a temporary class. The temporary class will be removed later.
2. The batch data structure and row-based layout can also be based on VectorizedColumnBatch and ColumnarRow, respectively.

Performance impact:

There will be *performance degradation* in this step due to the extra buffering and copying of the intermediate data. However, users can turn off such features to avoid performance degradation.

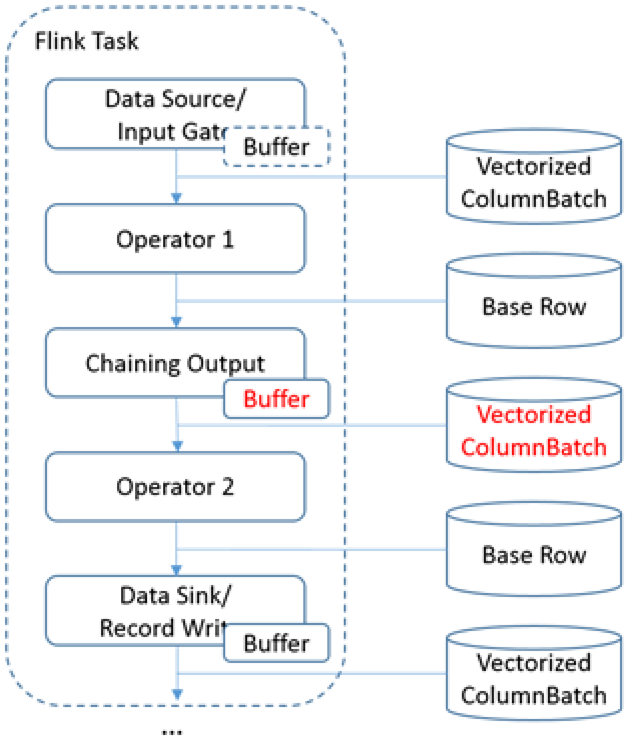

Fig. 5

5.2.4 Step 4 – Supporting Vectorization for Part of the Operators

In this step, we vectorize operators so that they take batches as input and produce batches as output. Since there are a large number of operators, it is difficult to vectorize them all at once. So we support them one by one, and allow row-based and vectorized operators mixed in an operator chain. For example, in Fig. 6, Operator 1 is vectorized while Operator 2 is row-based, and they are in the same operator chain.

The changes to an operator should be straightforward. For example, if the *processElement* of an operator looks like this:

```
public void processElement(StreamRecord record) {
        BaseRow row = (BaseRow) record.getValue();
        BaseRow newRow = new GenericRow();
        …
        output.collect(newRow);
}
```

The vectorized code is primarily based on batch-based processing (see Section 3):

```
public void processElement(StreamRecord record) {
        VectorizedColumnBatch batch = (VectorizedColumnBatch) record.getValue();
        for (int i = 0; i < batch.getSize(); i++) {
                BaseRow row = (BaseRow) batch.getRow(i);
                BaseRow newRow = new GenericRow();
                …
                output.collect(newRow);
        }
}
```

Please note that, if an operator is vectorized, its output is also a batch. So the buffer in the ChaingOutput can also be removed.

Implementation Concerns:

1. There is no need to change the interface of operators, since we can wrap the batch in the StreamRecord.
2. We should provide more subclasses for the ChaingOutput, to deal with the cases where:
   a) The upstream operator is vectorized while the downstream is row-based.
   b) The upstream operator is row-based while the downstream is vectorized.

Performance impact:

There should be performance improvements in this step, due to:
1. For vectroized operators, there is no need to buffer row data in the ChainingOutput.
2. Since the new operator processes multiple records together, JIT can make more optimizations, like SIMD.
3. In row-based processing, each row will go through the whole call stacks. In vectorized processing, however, multiple records amortizes the costs of the call stack.

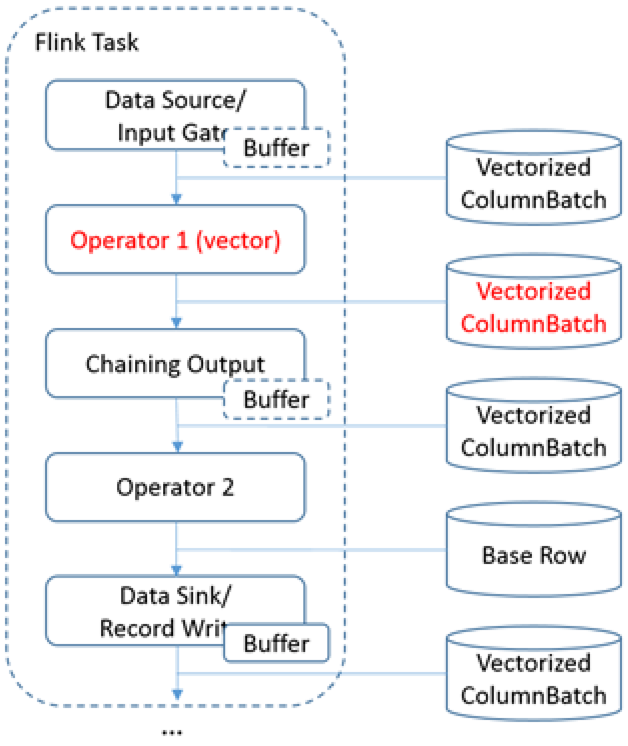

Fig. 6

5.2.5 Step 5 – Supporting Vectorization for All Operators

In this step, we finish the vectorization for all operators, so that all operators will take batches as input, and produce batches as output.

Implementation Concerns:

1. Since all operator outputs are batches, the buffer in ChainingOutput can be removed completely.
2. Similarly, the buffer in data sink/record writer can also be removed.

Performance impact:

1. There will be further performance improvements, as more operators are vectorized and more buffers are removed.
2. However, as discussed in Section 3, there can be cases where performance will degrade by vectorization. We need to use methods in Section 3 for such cases.

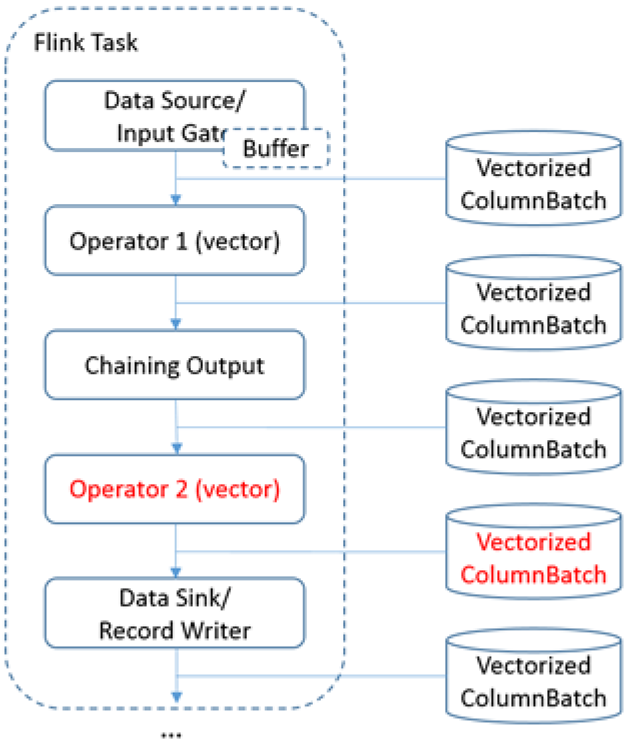

Fig. 7

5.2.6 Step 6 – Enhancing Vectorized Data Structures/Algorithms

So far, the basic vectorization framework is established. We will improve the data structures and algorithms for better performance. This is an open issue, and we invite the community to work on this with us.

Implementation Concerns:

For now, we can think of these issues:

1. Support batch/vector with Apache Arrow. The benefits include:
    a) Arrow is a standard data format, and adopting Arrow as the underlying data format can greatly facilitate data exchange.
    b) Arrow provides high-performance utilities to manipulate columnar data.
    c) We have contributed to Apache Arrow to make it easier to support Flink use cases.
2. Support vectorized hash tables for hash aggregate/hash join. Implement new data structures and algorithms with better performance for columnar data.
3. Support compression algorithms for data shuffling to improve performance.

Performance impact:

Performance will be further improved.

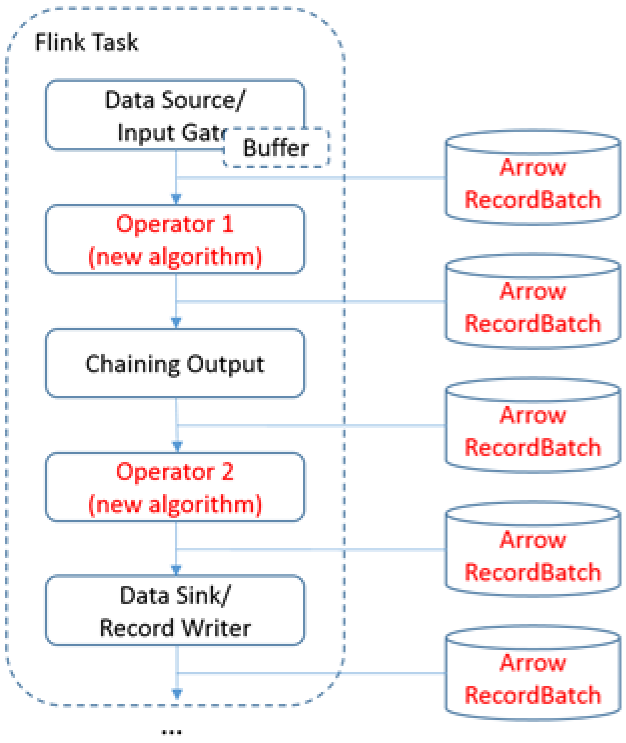

Fig. 8

5.2.7 Summary – Performance Impact of Each Step

According to the above descriptions, enabling vectorization can be achieved by the 6 steps. The performance impact of each step is summarized as follows:

Table 2

Step	Performance Impact
Step 1	No significant performance change
Step 2	No significant performance change
Step 3	Performance degradation
Step 4	Performance improvement
Step 5	Performance improvement
Step 6	Performance improvement

5.3 Additional Discussions

In this section, we give more discussions about implementation details. In particular, we discuss the following questions:

5.3.1 How to Determine the Batch Size

The batch size can be the performance key: If the batch size is too large, it may cause too much delay. In addition, a large batch would be unable to fit into the CPU cache, leading to more cache misses. In contrast, an overly small batch size would degenerate to row-based processing, which would lose the benefits of vectorization.

We determine the batch size according to the following principles:

1. The batch size cannot exceed a predefined max value. According to [8], the max value can be set around 1024.
2. All records in the same network/file system buffer can be split into a number of consecutive batches.
    a) Example 1: 2000 records in the buffer will be split into 2 batches, with 1024 and 976 records, respectively.
    b) Example 2: 3000 records in the buffer will be split into 3 batches, with 1024, 1024, and 952 records, respectively.
    c) Example 3: 500 records in the buffer will be treated as a single batch with 500 records.
3. Records in adjacent network/file system buffers can be merged, only if the time interval between them is smaller than a predefined threshold. This principle is to reduce the latency.
    a) Example 1: If the first buffer has 2000 records, it will result in two batches with 1024 and 976 records. If the second buffer comes within the interval threshold, then the first 48 records from the second buffer will be merged with the second batch of the first buffer to construct a batch with 1024 records.
    b) Example 2: In the above example, if the second buffer comes after the interval threshold, then the second batch from the first batch with 976 records will be sent as a separate batch, without merging records from the second buffer.

5.3.2 How to Adapt to the Mailbox Threading Model

The mailbox threading model [9] is the new threading model for stream tasks. It greatly simplifies the concurrent access of task states. For vectorization, the biggest challenge comes from the fact that, processing a batch may take much longer time (compared with processing a row), and during this time, the task would be unresponsive to other (possibly more urgent) events.

It should be noted that this problem also exists for row-based processing:

1. For some cases, processing a single row can also be time-consuming
2. The endInput method for some operators can be time consuming. For example, the endInput method for SortLimitOperator performs a complete sort of all records.

We provide two potential solutions for this problem:

1. Setup separate thread pools for ordinary actions and long-running actions. This violates the principle of the mailbox model that all actions of a task are processed sequentially in a single thread. So this solution only applies to operators that do not change task states.
2. Preemptive scheduling: A time-consuming action can be preempted, to give way to more urgent actions. When an action is preempted, it should insert another action to the queue, which contains the batch to be processed, as well as the index of the next record to process.

5.3.3 How to Accommodate Dynamic Input Selection

Dynamic input selection [10] is a new feature for operator processing. For this technique, two or multiple ways of the inputs can be switched even after a single record processing. This should be less of a problem for batch processing, since the decision of input switch is made once for each batch. This behavior is in fact consistent with the looser contract of InputSelection (introduced by Piotr Nowojski):

l For Hash Join, there is no problem, since the switch only happens when the build side is finished.
l For Merge Join, an extra buffer is required, and the record batch acts as the buffer.

Therefore, there is no need to change the operator code.

References

[1] DB2 with BLU acceleration: So much more than just a column store. VLDB, 2013
[2] SQL Server column store indexes. SIGMOD, 2011
[3] MonetDB/X100: Hyper-pipelining query execution. CIDR, 2005
[4] Exploring query compilation strategies for JIT, vectorization and SIMD. IMDM, 2017
[5] JAVA and SIMD. http://prestodb.rocks/code/simd/
[6] ColumnStores vs. RowStores: How Different Are They Really? SIGMOD, 2008.
[7] Integrating Compression and Execution in Column-Oriented Database Systems. SIGMOD, 2006.
[8] Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. VLDB, 2018.
[9] Change threading-model in StreamTask to a mailbox-based approach.
https://docs.google.com/document/d/1eDpsUKv2FqwZiS1Pm6gYO5eFHScBHfULKmH1-ZEWB4g
[10] Enhance Operator API to Support Dynamically Selective Reading and EndOfInput Event.
https://docs.google.com/document/d/10k5pQm3SkMiK5Zn1iFDqhQnzjQTLF0Vtcbc8poB4_c8/edit#heading=h.fvegws3z4g3h