Enorm: Efficient Window-Based Computation in Large-Scale Distributed Stream Processing Systems

Kasper Grud Skat Madsen Yongluan Zhou Li Su

University of Southern Denmark
{kaspergsm, zhou, lsu}@imada.sdu.dk

ABSTRACT

Modern distributed stream processing systems (DSPS), such as Storm, typically provide a flexible programming model, where computation is specified as complicated UDFs and data is opaque to the system. While such a programming framework provides very high flexibility to the developers, it does not provide much semantic information to the system and hence it is hard to perform optimizations that has already been proved very effective in conventional stream systems. Examples include sharing computation among overlapping windows, co-partitioning operators to save communication overhead and efficient state migration during load balancing. In lieu of these challenges, we propose a new framework, which is designed to expose sufficient semantic information of the applications to enable the aforementioned effective optimizations, while on the other hand, maintaining the flexibility of Storm's original programming framework. Furthermore, we present new optimization algorithms to minimize the communication cost and state migration overhead for dynamic load balancing. We implement our framework on top of Storm and run an extensive experimental study to verify its effectiveness.

CCS Concepts

•**Information systems** → **MapReduce-based systems;**

Keywords

Adaptation, Resource Management, Programming Model

1. INTRODUCTION

There is an emerging interest in building next-generation large-scale distributed stream processing systems (DSPS), such as Storm [30], MillWheel [1] and Spark Streaming [32], which make use of large-scale computing clusters to perform continuous computations over fast streaming data. To support complex application logic that is not easy to express in a declarative language, such as matrix multiplication and various data analytics algorithms, these systems support complex user-defined functions (UDF) implemented using imperative programming languages and a simple data model, often opaque to the system.

We observe that, assuming the semantics of UDFs and data are opaque to the system, makes it hard to employ some important optimizations that have been proved to be effective in conventional SQL-like data stream systems. First of all, window-based computation, which continuously executes functions over sliding windows, is a fundamental type of computation for data stream applications. In a DSPS, a job contains a complex topology of operators and each operator would be parallelized onto a large number of asynchronous instances, which renders the maintenance of the advancement of the sliding windows challenging. Furthermore, a job in a DSPS may perform similar computation on overlapping windows that share a lot of common data. Sharing the computation among overlapping windows is known to be an effective optimization technique in conventional data stream systems [18, 5], which unfortunately is not employed by the state-of-art DSPSs.

Another effective optimization for DSPS is to collocate the parallelized instances that communicate extensively between each other [17, 12], which would not only reduce the bandwidth consumption, but also save the CPU overhead for data serialization and deserialization. The effect of such an optimization depends on how the operators are parallelized. For example, if the input of two adjacent operators can be partitioned in a way such that each instance of the upstream operator only communicates with one instance of the neighboring operator, and vice versa, then we can completely eliminate the communication between these two operators by collocating every pair of communicating instances onto the same node. However, in existing DSPSs, inputs of an operator are partitioned based on the key of its input, which is assumed to be an opaque blob. Therefore, the system does not have the opportunity to adjust the partitioning key to exploit this kind of optimization.

Last but not the least, dynamic load balancing is important to maintain low processing latency under significant runtime load variation in a long-standing stream job. Load balancing requires moving tasks from one node to another. For a stateful operator, e.g. a window-based operator, moving its tasks would incur the movement of computation states. To reduce the cost of state migration, more semantic information of window-based computation is needed.

DEBS '16, June 20 - 24, 2016, Irvine, CA, USA

ISBN 978-1-4503-4021-2/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2933267.2933315

In this paper, we intend to solve these issues by building a programming framework on top of Storm, which still keeps most of the flexibility of the original Storm framework, but at the same time achieve high efficiency in window-based computation and low overhead in both data communication and dynamic load balancing. In summary, the contributions of this paper include the following.

- We propose a framework, which natively supports window management. Our framework not only simplifies the development of complicated window-based computations, but also enables transparent sharing of computations on overlapping windows.
- By extending the opaque partitioning "key" to a set of named attributes, our framework can obtain more information about how the input of operators are partitioned. Therefore, it can check if two or more operators can be parallelized consistently.
- We propose an optimization algorithm to minimize the communication cost by grouping consistent operators into components. By parallelizing the operators in each component using the same set of attributes, we can eliminate the communication across these operators.
- We also propose efficient task migration techniques which utilize the additional semantic information available in our framework to minimize the overhead of load balancing.
- We implement our framework on top of Apache Storm and perform extensive experiments on Amazon EC2, which verify the effectiveness of our techniques.

2. MOTIVATING EXAMPLE

We now provide a motivating example to investigate what desireable properties a general DSPS framework should provide, and give some intuition into how our framework does so.

Description. The job shown in figure 1 consists of 5 operators, which is used to make various computations for a peer-to-peer lending company. Within this job, op_1 calculates the interest rate per incoming data tuple; op_2 calculates the interest rate vs. the annual income per area and op_3 calculates the approval rate of loans per area; op_4 calculates the globally average interest rate and op_5 calculates the global approval rate.

Assume the schema of the input data is: $\langle$*user, area, date, approval, amount, return, lendingTime, income*$\rangle$. Operator op_1 is partitioned on all these attributes, while op_2 and op_3 are partitioned only on the *area* attribute. op_4 and op_5 cannot be parallelized, as they calculate global values.

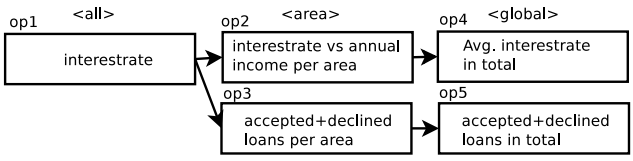

Figure 1: Motivational Job

Issue 1. Executing the job, potentially requires output from op_1 to be sent over the network to both op_2 and op_3, as each instance of op_2 and op_3 takes the outputs from a subset of op_1 instances as their inputs.

Optimization 1. Transparently detect and group compatible operators to minimize the overhead of inter-operator communication. The solution for the example job is seen in figure 2. Fused operators are called components, in this case, operators op_1, op_2 and op_3 are grouped as component C_1 and partitioned on the $\langle area \rangle$ attribute.

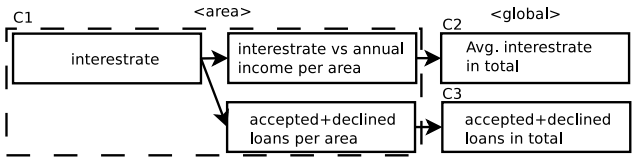

Figure 2: Motivational Job w. Components

Issue 2. Operators op_2, op_3, op_4, op_5 need to maintain windows with overlapping computation. Assume they must produce an output each 10 minutes, 30 minutes, 1 hour, 6 hours and 24 hours. This requires the end-user to implement a window management mechanism to maintain the states for multiple windows concurrently.

Optimization 2. Our framework can compute the windows, by either executing each input once for each overlapping window or once per tuple to form partial results, which are then merged by the consolidate operator. The decision can be made a runtime time, based on a cost-model [23], as long as the job is prepared to use the consolidate operator. We now describe how this is done for the motivational job.

As presented in figure 3, the operators op_2 and op_3 are each split into two operators, (1) a compute operator, which calculates partial results for 10-minute intervals and (2) a consolidate operator, which merges the results of 10-minute intervals, in order to produce outputs for the user-specified window intervals. This means the compute operator is executed once for each incoming tuple and the consolidate operator is executed once for each required output.

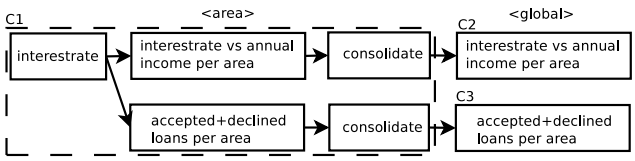

Figure 3: Motivational Job w. Components & CS

Issue 3. Rebalancing instances among nodes, requires migrating instances from overloaded nodes to those which are capable of taking over the computation. If the situation happens frequently, the migration operations can incur extensive processing latency.

Optimization 3. We noticed that, compared to compute operators, the workloads of consolidate operators are much more stable, as they only execute results from compute operators, which usually have a fixed rate. This property means that the probability of rebalancing a consolidate operator is much less than that of compute operators. Therefore, our framework supports merging the consolidate operators into the downstream neighboring components at runtime, as presented in figure 4. After this merge, there are only compute operators within component C_1. By updating the input par-

titioner of C_1, one can migrate an instance of C_1 from one node to another. The migration of a compute instance can be done even without moving its state. For instance, for a 10-minute window interval, suppose the results of the first 5-minute interval are calculated on node A and the results of the second 5-minute interval are calculated on node B after migration, the downstream consolidate operator can merge the partial results from node A and B to generate the results of the complete 10-minute window instance. Notice also that the extra overhead of merging partial results is only temporary, because the results from the "old" nodes (node A in this example) will eventually no longer be needed, once they are not included within any future user-specified window intervals.

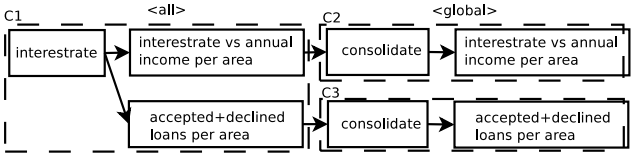

Figure 4: Motivational Job (split)

Benefits. As is clear from the above discussion, there are large benefits when working at a higher level abstraction with clearer semantical meaning, which allows the following advantages of our framework:

1. Transparent creation of components (fusion).
2. Transparent window management.
3. Transparent sharing of computation between overlapping windows (used only when most efficient).
4. Efficient state migration.

3. RELATED WORK

3.1 Programming Models

The MapReduce framework [10] is designed for batch processing, where data is known beforehand and significant processing latencies are acceptable. The programming model of the MapReduce framework can be modified [7], such that the framework becomes suitable for low-latency processing. In short, the reducer can be modified to do eager processing of inputs and produce preliminary output. This has lead to several similar low-latency programming models, such as those employed by Apache Storm [30], Apache S4 [25], Google MillWheel [1] and more [20, 9]. Our approach is an extension of such typical low-latency programming models, which employs an additional operatortype to allow a large set of optimizations.

3.2 Runtime Load Balance

Adaptation. One of the most common ways to repartition data at runtime, is to make global decisions based on statistics collected at runtime, and then choose the most appropriate state migration techniques to effectuate the decisions [14, 27, 13, 19]. ESC [27] is such a system, which supports scaling and load balancing by using an Autonomic Manager, which has access to information about the workload of each node and the queue lengths of the worker processes, to make global decisions.

In this work, we use Flux [28] to make decisions on which workloads to migrate between nodes. Flux works periodically as follows. It sorts nodes in descending order of their workloads. Then it moves the biggest suitable data partition at the first node to the last one in the sorted list, such that load variance is decreased. If necessary, it also moves the biggest suitable data partition at the second node to the second last one in the list, and so on.

There are many different variations of load balancing, such as those proposed by [34, 33, 22].

State Migration. The simplest state migration technique is called *direct state migration*. It works by first pausing the processing of the workload to migrate, then serializing the state and sending it over the network, before unpausing the processing again. The latency therefore primarily depends on the statesize [21].

In some circumstances it is possible to improve the state migration efficiency, by exploiting certain features of a DSPS [29]. A good example of this is checkpoint-assisted state migration [21, 8] in DSPSs which are executed with passive fault-tolerance [16, 6]. The idea is that a checkpoint already contains state, and if migrating a workload to a node with a recent checkpoint, it is possible to save the cost of serializing the state (i.e. direct state migration) and only incur the cost of replaying buffered tuples from the passive fault-tolerance.

The authors in [2] discuss how to avoid the need for costly state migration, by adding an operator to perform aggregation of the partial results of the scaled operators. A recent technique called "The Power of Two Choices" (PoTC) [4], extends this idea, by defining two hash functions $h_1(x)$ and $h_2(x)$, such that each key x can be sent to one of two alternative downstream operator instances. Each operator instance balances the amount of work sent downstream, such that all operator instances receives an even workload. Since the state is split over two operator instances per key, the partial states must be merged before the final computation can be applied. The same authors [24] then studied scenarios where two choices are not enough to obtain sufficient load balance. They identify hot operator instances, i.e. those which are more heavily loaded than the rest and allow these to be executed on more than two nodes. They show this allows the system to obtain a very good load balance. The overhead of their techniques can become large, because hot keys can have more duplicated state, which leads to more required merges.

Our framework extends on the idea of maintaining partial states. With the help of the *consolidate* operator, the framework avoids the continuous overhead as e.g. the PoTC approach suffers from.

Collocation. Operator instances which are exhibiting extensive direct communication, can preferably be collocated on the same nodes, in order to minimize the inter-node communication. The following papers all tries to solve the problem of minimizing the inter-node communication, while ensuring each node is loaded less than some user-defined value.

The authors in [3] defines the *traffic-based scheduler*, which is a heuristic approach relying on runtime statistics. The solution is recalculated and effectuated at runtime, if the load of any node becomes too large, or it is possible to reduce the inter-node communication by some user-defined percentage. The authors does not consider the overhead of their approach.

The authors in [31] defines a traffic-aware scheduler. Their approach can incur an extremely high overhead if stateful computation is employed, since the overhead of the approach is then unbounded. Fischer et al [11] defines a solution which models the objective as a graph partitioning problem. They consider only cpu load. Peng et al [26] claims the problem is infeasible to solve optimally, and then process to define a heuristic solution. Their approach considers both cpu, memory and bandwidth.

In this work, we employ a heuristic approach to minimize inter-node communication at job-submission time, with the help of operator fusing. This allows the framework to support standard techniques for load balancing, which would otherwise nullify the benefits of collocation over time.

3.3 Sharing Computation

In DSPSs with native support for windows, it is possible to share computation between overlapping windows, as discussed in [18]. It is possible to share computation for different classes of aggregation functions, different window typic and different input models [5]. A short paper [23] provides a cost-model for determining if sharing computation between overlapping windows is beneficial.

4. FRAMEWORK

4.1 Data Model

Data is modeled as a number of continuous streams of tuples in the form of $\langle key, value, ts_1, ts_2 \rangle$.

- The *key* is an n-tuple, containing a set of partitioning attributes. It is the key that decides how the tuple is routed.
- The *value* is an n-tuple, containing user-defined values.
- The ts_1 and ts_2 are timestamps associated with the tuple, which is used to indicate what timespan the tuple belongs to. This is especially useful when considering time-based windows, as a tuple might represent a value calculated over a specific window instance.

4.2 Programming Model

Enorm supports hopping, tumbling and overlapping time-based windows, defined by a starting time, a length and a frequency (of window instances). It is up to the end-user to decide how to handle data that spans multiple windows, which is also defined as part of a window specification. Enorm uses a slack s to determine when all data for a window $[ts_i, ts_j]$ is received at a given operator instance, by ensuring a punctuation with timestamp $t \geq ts_j + s$ has been received from each upstream operator instance. Punctuations are periodically sent from the input operators (spouts in Apache Storm terminology) and emitted downstream on the same communication channels as the data. Enorm supports sharing of computation between overlapping windows, by transparently maintaining state in partials of windows, and later merging the partial states into the user-defined complete windows. Enorm uses a cost-model to detect if it is cheaper to do this merge, compared to calculating duplicate state as typically done.

The MapReduce framework [10] specifies two main operators, *map* and *reduce*. The *map* operator supports stateless low-latency processing and can therefore be used directly in our framework. The *reduce* operator is split into two operators, called *compute* and *consolidate*. Our framework consists of the operators:

$map(k_1, v_1, t_1, t_2) \rightarrow list(k_2, v_2, t_3, t_4)$, takes one tuple and eagerly outputs a list of new tuples, by executing the user-defined logic in the function. It does not support state.

$compute(k_2, v_2, t_3, t_4) \rightarrow (k_2, v_3, t_5, t_6)$, takes one tuple and eagerly process it to build the state for the corresponding window (or a partial of the window). The compute function makes memory storage (state) available to the end-user, by exposing a map from a key to a user-defined object. The user can store any state within this structure. After all input for a given window (or partial of a window) has been processed, the state is sent to the downstream operators, such that all values for a given key, is sent to the same instance of a given downstream operator.

$consolidate(k_2, list(v_3), t_5, t_6) \rightarrow list(k_3, v_4, t_5, t_6)$, takes a list of partial windows (with partial states) and merges them into complete windows as per the window specification. The user must specify the logic needed to merge partial windows. The operator will only output tuples when results for a completed window is created.

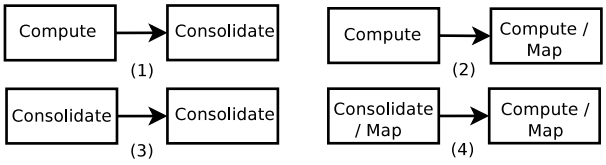

Figure 5: Operator Configurations

The functionality of the operators, depends on their configuration. Figure 5 shows the possible configurations.

Figure 5.1 shows a compute operator followed by consolidate. The consolidate operator maintains one or more time-based windows, which are calculated by either (1) merging partial results from upstream compute or (2) computing state for each overlapping window. The actual technique is determined at runtime. The compute operator processes incoming tuples in an eager fashion. In case the cost-model defines that it is most efficient to use automatic sharing of computation between overlapping windows, the compute function will only output partial results, which are then merged by the consolidate function. Otherwise, the compute function will compute distinct output for each window, by executing the user-logic for each window and sending tuples downstream when a window instance is completed. In this case, the consolidate operator just checks for any partial data to merge, which are the results of state migration, before sending along. Details on state migration will be explained later on.

Figure 5.2 shows a compute operator which is sending data to a compute (or a map). In this case, the upstream compute operator will not produce partial results, but instead do regular eager processing of tuples. This configuration does not support automatic sharing of computation between overlapping windows, nor does it support migrating instances of the compute operator without performing state migration.

Figure 5.3 shows a consolidate operator sending its output to another consolidate operator. The upstream operator produces outputs for complete window instances, which are simply considered as partial results by the downstream con-

solidate operator, whose logic is similar to the one in figure 5.1.

Figure 5.4 shows the simplest case, as tuples are sent along without any special logic. That is, the upstream operator will output tuples as soon as they are available and the downstream operator will eagerly process them. The difference between compute and map in this configuration is that compute maintains a key-value store, while map does not.

4.3 Operator Model

A DSPS is executing a set of long-standing queries, which can be represented as a DAG $\langle O, E\rangle$, where each vertex is an operator O_i and the direction of the edges represent the direction of data flow. Each operator O_i is characterized by a tuple $\langle IS(O_i),\ OS(O_i),\ PA(O_i),\ List(S(O_i))\rangle$.

IS_i and OS_i are the input and output data schema(s) and $PA(O_i)$, called *partition attributes*, is a subset of attributes in $IS(O_i)$, which will be used to parallelize O_i. Each unique combination of the values of attributes in $PA(O_i)$ defines a unique partition of O_i.

In a parallelized query plan, an operator O_i could be partitioned onto a number of computing nodes according to $PA(O_i)$ and the maximum number of partitions that we can generate using $PA(O_i)$ is called the partition cardinality of $PA(O_i)$, which is denoted as $PA^c(O_i)$. In this paper, we assume every tuple can be uniquely identified with all of its attribute values. So if $PA(O_i) = IS(O_i)$, then we denote $PA^c(O_i) = \infty$, which means that the number of partitions of O_i is only limited by the cardinality of the input which is unbounded in a data stream system. Another boundary case is that $PA(O_i) = \varnothing$ and in this case, $PA^c(O_i) = 1$.

Example 1a. Suppose we have an operator O_1 which is counting words and has input schema $IS(O_1) = \langle word\rangle$ and output schema $OS(O_1) = \langle word, wordcount\rangle$. Say we have another operator O_2 which is counting letters of words, with input schema $IS(O_2) = \langle word\rangle$ and output schema $OS(O_2) = \langle letter, lettercount\rangle$.

If O_1 and O_2 are partitioned on the attribute word, we get $PA(O_1) = PA(O_2) = \langle word\rangle$. The maximal partitions that can be created for both operators is equal to the number of distinct words in the input dataset.

Example 2a. Lets return to the motivational example depicted in figure 1. Operator 1 has $PA(O_1) = \langle all\rangle$, with $PA^c(O_1) = \infty$. Operator 2 has $PA(O_2) = \langle area\rangle$, with $PA^c(O_2) = |area|$. Operator 4 has $PA(O_4) = \varnothing$ with $PA^c(O_4) = 1$. Operator 3 is the same as operator 2 and operator 5 is the same as operator 4.

A stateless operator is defined as an operator which does not maintain any computation state. The consolidate operator is stateless, even though it buffers data, as any lost state (from instance migration without state), will simply be resent from the upstream operators.

A stateful operator maintains states while executing the incoming tuples. The compute is the only stateful operator in the framework.

A flexible operator is oblivious to how incoming data is partitioned, which means the partitioner can be changed without doing state migration.

A rigid operator requires that states must be redistributed when modifying the partitioner, to ensure tuples are executed based on correct states. See table 1 for an overview of how the map, compute and consolidate operators are categorized.

Operator	Properties
Map	Stateless and Flexible
Compute	Stateful and Flexible
Consolidate	Stateless and Rigid

Table 1: Overview of operators

4.4 Parallel Query Processing

Fusion is the act of combining operators into units called components. We define $PA(C_i) = \cap_{(O_j \in C)} PA(O_j) \neq \varnothing$ and a component C_i can at most be parallelized onto $PA^c(C_i)$ nodes. A component is stateless, if it contains only stateless operators and a component is flexible, if it contains only flexible operators.

Example 1b. Continuing with the previous example 1a (section 4.3), the operators O_1 and O_2, can be combined as component C_1, with $PA(C_1) = \langle word\rangle$, and the maximal number of partitions which can be generated for C_1 is in this case $PA^c(C_1) = PA^c(O_1) = PA^c(O_2)$.

Example 2b. For the motivational job given in figure 2, component 1 has $PA(C_1) = \langle area\rangle$, with $PA^c(C_1) = |area|$ and component 3 and 4 has $PA(C_3) = PA(C_4) = \varnothing$ with $PA^c(C_3) = PA^c(C_4) = 1$. Compared to the original job, the transformed one has less potential for scaling, but benefits from faster execution.

4.4.1 Fusion

The process of constructing components is called *Fusion*, which can either be specified by users manually or be done by automatic recognition. We now describe how to automatically fuse operators. The objective of fusion is to minimize the inter-component communication, such that:

1. Each operator O_j belongs to exactly one component
2. $\forall_{C_i} : \cap_{O_j \in C_i} PA(O_j) \neq \emptyset$

Load balancing is not considered at this point, because the load of each component can be tweaked by adjusting the number of instances which are initialized for a component. A node can execute many instances of a given component, each processing a subset of the inputs.

Algorithm 1 presents a greedy algorithm, which can fuse operators. The algorithm works greedily by merging the operators with largest score, as long as they are partitioned in a compatible way. The score is the rate of data tuples transferred between the operators to fuse. Given the input from figure 1, algorithm 1 returns the result as shown in figure 2. Algorithm 1 requires information about the rate of data transmission between all operators, which usually requires a trial execution of the job. In some cases, this might be inconvenient (or the input rate might be very unstable), for which reason we here define an alternative scoring mechanism whose objective is to minimize the number of components.

Alternative Scoring. Minimizing the number of components indirectly minimizes the inter-component communication. The alternative score is based on the commonness of the partition attributes between operators which are not fused. The score of fusing operators op_i and op_j is calculated by taking the intersection of the partitioning attributes and summing the values. The reasoning behind fusing operators which have most partitioning attributes in common, is to

Algorithm 1: Automatic Fusion

```
Input: operators O, partition attributes per operator,
       output rate between any pair of operators
opFused ← newMap()
for each o ∈ O do
    opFused(o) ← {o}
while true do
    maxScore ← 0
    for each i ∈ O do
        srcComponent ← opFused(i)
        srcPA ← getPA(srcComponent)
        for each j ∈ i.downstreamOP do
            if srcComponent contains j then
                continue
            dstComponent ← opFused(j)
            dstPA ← getPA(dstComponent)
            if size(intersect(srcPA, dstPA)) = 0 then
                continue
            score ← getOutputRate(i, j)
            if score > maxScore then
                maxScore ← score
                src ← i
                dst ← j
    if maxScore > 0 then
        fuse src and dst, such that src and dst points to
        the same component in opFused
    else
        break
return opFused
```

maximize the probability that more fusion operations can be performed afterwards.

4.4.2 Key Expansion

If input data to flexible operators are randomly partitioned, significant key expansion may happen, that is, each key in the key space is maintained on each node of the flexible operator. This leads to (1) worse processing latency, since building new state is more time-consuming than updating existing state and (2) excessive memory utilization and (3) increased overhead of consolidating the partial results, whose size increase with key expansion.

There are multiple ways to handle this problem. One recent approach is called "Power of Two Choices" (PoTC) [4], which ensures each input can be processed by maximally two different nodes. This approach will result in a reasonably load-balanced operator, but will maintain up to two times of the number of keys and associated values, and also incur a continuous overhead for merging partial results.

Our framework takes another approach to avoid the continuous overhead of key expansion. According to the data model, each tuple is associated with a key. These input keys can be partitioned into a number of non-overlapping subsets, each called a component instance or a key group. Each key group is independent of one another and each can maintain a separate processing state if applicable. A partitioner can now be defined by a hash function, which maps an input tuple to a key group. Using this logic, key expansion no longer exists as each key is sent to only one node. Problems with overload, underload and skewness can be handled by changing the location of a subset of key groups, such that they are to be executed on a new set of nodes. This can be done instantly, without moving any state, and will only result in a *temporary* key expansion, i.e. state is only maintained on two nodes while the state on the "old" node is still valid.

Choosing which key groups to migrate is the job of an adaptation algorithm. In this work, we employ Flux[28], though our framework can support any kind of adaptation algorithm.

4.5 Adaptation Model

We now describe how adaptation operations are modelled.

Adaptation Categories. Adaptation strategies can be grouped into three categories:

1. *Instant migration.* With this strategy, the migration does not involve movement of the state of the running instance and the new operator instance can be fired up at the destination node (almost) instantly. This is applicable when the operator to be migrated is stateless or flexible.

2. *Disruptive migration.* This strategy takes a pause-and-migrate approach, where we first pause the processing of the relevant operator instances and move the their states to the destination nodes. The input data will be redirected from the old instance to the new instance of the operator. After the migration, the old instance will be scraped while the new one will resume the processing based on the migrated state. Note that, in this strategy, both the old and new instances of the operators cannot execute any input data during the migration.

3. *Smooth migration.* This strategy can be applied to perform migration of window-based operators that conducts computations over a sliding window. In general, this strategy keeps the old instance of the operator running while it initiates the new instance at the destination. The old instance will keep generating the results for the current window instance while the new instance will generate results for the next window instance, whose states are not overlapped. Notice that for overlapping windows, this will result in tuple duplication while the adaptation operation is being effectuated.

Basic Adaptation Operations. The adaptation operations discussed in this paper are defined as follows:

1. $Migrate(O_i, C_j, C_k)$:

   (a) migrate O_i from C_j to C_k so that $C'_j = C_j \setminus \{O_i\}$ and $C'_k = C_k \cup \{O_i\}$.

   (b) $PA(C'_k) \leftarrow PA(C_k) \cap PA(O_i)$

   (c) $PA(C'_j) \leftarrow \cap_{(O_l \in (C_j \setminus \{O_i\}))} PA(O_l)$

   In practice, this adaptation operation is useful under the following two circumstances: (1) $PA^c(C_j)$ is too small and C_j cannot be scaled sufficiently out; (2) $PA(C'_j) = IS(C_j)$, which means the new component C'_j can be partitioned arbitrarily and hence it can be re-scaled with minimum cost. This operation is what is applied in the motivational example, when the job is transferred from figure 3 to figure 4.

2. *RePartition*(C_i, N_j, N_k) Given the current partitioning scheme of C_i on a set of nodes N_j, this operation re-allocates the partitions of C_i to a new set of computing nodes N_k. If N_k is equivalent to N_j, then this operation can be used to rebalance the load distribution among N_j. Otherwise if N_k is a superset (or subset) of N_j, then the operation is to increase (or reduce) the resource allocated to the processing of C_i.

The model is flexible and general enough to allow users to specify both complicated and simple adaptation techniques with ease. Consider for instance the direct state migration, which is of type *Disruptive Migration* and can be expressed with the *RePartition* operation.

Composite Operations. The basic operations can be used to form composite operations which are carried out on the components. For example, the operation of allocating one more computing node n_l to process component C_i could be composed by the following sequence of basic adaptation operations: *Migrate*$(O_k, C_i, C_j) \rightarrow$ *Repartition*(C_i, N_i, N_i'), where $N_i' = N_i \cup \{n_l\}$.

5. ADAPTATION

In this section, we describe how to efficiently migrate the instances of components in our framework.

Scaling a Flexible Stateless Component. The component can only consist of map operators. Since the component is flexible and stateless, it is straight-forward to adapt, using the strategy of *Instant Migration*. A *shuffle* partitioner can be used to reallocate the key group, which also means the component can be balanced without performing any state migration operation.

Scaling a Flexible Stateful Component. The component can consist of map and compute operators, and can be migrated using *Instant Migration*, *Disruptive Migration* and even *Smooth Migration* if the component conducts windowed computation. From previous works [21], it is known the latency of disruptive migration depends heavily on the size of the migrated state, and the completion time of smooth migration depends heavily on the window length.

We now discuss the implications of using *Instant Migration*, which is applicable when the downstream component first applies a consolidate operator to merge partial results. Consider that by changing the location of key groups, without migrating any state, partial state for a key will exist on multiple nodes. This means the migration increases the number of keys to consolidate on the downstream operators (key explosion, see section 4.4.2). The implication is that more data must be serialized, sent over the network, deserialized and consolidated. This side-effect lasts until the state on the node which initially contained the migrated key group becomes useless. The benefit of this approach is that it does not incur any latency of state migration and can be applied instantly.

Scaling a Rigid Stateless Component. The component can consist of map and consolidate operators. Remember that consolidate operators do not need to maintain state, but instead require that any missing state is resent from their upstream components. It is therefore possible to migrate this component without state, which is most efficiently handled using *Instant Migration*.

Scaling a Rigid Stateful Component. The component can consist of map, compute and consolidate operators. In case the component is windowed, migrating its instances can only be done using *Disruptive Migration* or *Smooth Migration*. Notice that this kind of component is the only one which cannot directly support *Instant Migration*.

Since instant migration is the most efficient migration strategy, we define a split operation which extracts certain operators from the rigid stateful component, thereby allowing them to be efficiently adapted. We also define the combine operation, which functions in the opposite way as the split.

5.1 Split & Combine

Split transforms a rigid stateful component to a flexible stateful component, by migrating the consolidate operators to the immediate downstream component. Figure 6 shows the scenario. In this work, we assume the immediate downstream component is either flexible or partitioned in a "compatible" fashion, i.e. $PA(C_x) \cap PA(C_{x+1}) \neq \emptyset$. The split operation can be handled with *Instant Migration*.

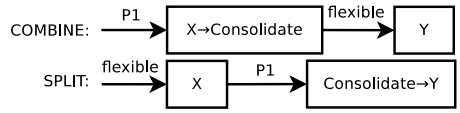

Figure 6: Split & Combine

Combine is the opposite operation of split, and can be applied using *Disruptive Migration* or *Smooth Migration* depending on the job. Combine is the only adaptation which cannot be done instantly. This is acceptable, as combine is needed only to improve the long-term performance, not to handle changes in load distributions fast.

One may have noticed that, except for the Combine operation, all the other adaptation operations considered so far can be handled using *Instant Migration*, which means adaptation operations in our framework are very efficient.

5.1.1 Cost-Model

In this section we give a cost-model to determine the cost in terms of cpu and latency, when executing either a Combine or Split operation.

Definitions. Let us denote the interval of the longest window for an operator op_i as p, the set of tuples to process within p as T_p and the average cost to execute the user-logic for a tuple as E. Let W be the number of overlapping windows and PW_p be the number of partial windows needed within p. The average number of migrated key groups (for load-balance) within p is denoted as MK_p and the cost of migrating a key group containing x copies of state is denoted as K_x. Lastly, let L be the average latency of migrating a key group with state.

Split - CPU Cost A split component executes each incoming tuple once (when forming partial windows), which gives a cost $T_p \cdot E$. A split component also incurs the cost of sending the results of partial windows from an upstream compute operator to a downstream consolidate operator, this part of cost is calculated as $PW_p \cdot K_1 \cdot \#keygroups$. As discussed previously, load migration leads to temporary extra state, which must also be consolidated, whose cost is $MK_p \cdot K_1$. In total, the CPU cost of split is $T_p \cdot E + PW_p \cdot K_1 \cdot \#keygroups + MK_p \cdot K_1$.

Split - Latency When the results of a partial window is to be serialized and sent to a downstream component, a

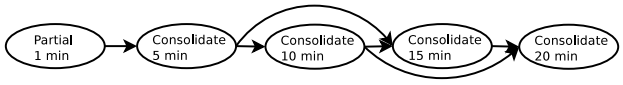

Figure 7: Efficient Consolidation

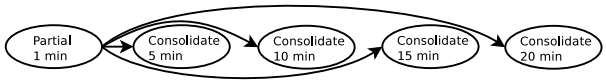

Figure 8: Naive Consolidation

separate thread can be used such that it does not impact the processing of newly arriving input data. This is because the results to be sent will not be changed anymore. The latency is therefore negligible, when assuming the load of the involved nodes is at a reasonable level.

Combine - CPU Cost The combined component must execute each incoming tuple multiple times, to build state for each overlapping window, which costs $T_p \cdot W \cdot E$. When migrating a combined component it will contain both the newest state and also partial windows for the consolidate. The cost is $MK_p \cdot K_{pw_p} = MK_p \cdot K_1 \cdot PW_p$. In total $T_p \cdot W \cdot E + MK_p \cdot K_1 \cdot PW_p$.

Combine - Latency The latency of migrations for load-balance is: $MK_p \cdot L$.

5.1.2 *Deciding when to split or combine*

We make decisions of split/combine components with the objective to minimize the CPU cost per component, while bounding the maximum latency to a user-defined threshold.

The combine operation can not be applied using the *Instant Migration*, which therefore can be quite costly. In addition, the properties that influence the costs and gains of conducting split/combine, such as the average number of state migrations and the average size of states, are expected to be changed slowly over time. We thus decide to periodically evaluate and apply the split/combine operations among components to control the overhead.

After every period p, we evaluate each component with the aforementioned cost model and apply split or combine if necessary. We always choose the most beneficial action, based on the objective described above.

6. IMPLEMENTATION

We have implemented our framework on top of Apache Storm.

6.1 Efficient Consolidation

Each consolidate operator can support multiple windows, which are calculated by iterating over all relevant partial results as defined in section 4. In case multiple windows are "compatible", meaning one complete window can be considered a partial result to another window, then these larger partial results can be used to form the new window. See figure 7 for an example, where one consolidate operator must form complete windows each 5, 10, 15 and 20 minutes, while receiving partial windows of one minute each.

To produce outputs for one hour, the efficient approach (figure 7) consolidates 86 partial results, while the naive approach (figure 8) consolidates 240 partial results.

There are multiple ways to form complete outputs with the minimum number of consolidates, consider for instance how the 20 minute window can be formed from 2x10 minutes or 1x5 and 1x15 minutes. The best choice is the one which has the smallest number of keys to consolidate, which can only be determined at runtime.

After a job is submitted, the framework builds a DAG, which describes all the ways to form complete results for each consolidate operator. The DAG is duplicated to all consolidate operators and the structure together with statistical information about the number of keys in each window (partial and complete) are used to make a decision on how to form complete windows at runtime.

6.2 Synchronizing Modifications

Our framework uses two kinds of punctuations; timestamp punctuations and modification punctuations. The modification punctuations are used to coordinate dynamic changes and the timestamp punctuations are used to detect when all expected data is available for processing a given window.

Our framework periodically outputs timestamp punctuations from the first components in the job, which contains the timestamp of the newest processed data minus a user-defined grace period, which is the maximum supported unorderedness of the data. When a component receives a timestamp punctuation, it stores it in a map, such that the timestamp for the specific connection is updated. Then the component calculates the oldest timestamp in the map and sends it along to all the downstream components. The window manager maintains a series of window instances per components, and all window instances which ends before the oldest timestamp in the aforementioned map can be processed, as all data must be available.

To support scaling operations, it is necessary to modify the logic of multiple components at runtime. Since each component is processing tuples in an unsynchronized fashion, there can easily be a different set of window instances on the different components. Each component classifies its window instances into one of three types: *active*, *processing* and *completed*. An active window, is one which is not ready for final processing, i.e. it is still missing data. A processing window, is ready for final processing and is either being processed or waiting to be. Lastly, a completed window has been fully processed. In order to synchronize updates at runtime, we employ a *modification punctuation*, which is guaranteed to modify the exact same set of *active* window instances on all components to be modified.

Punctuations are sent on the same queue as the data. Assume a modification punctuation mp_1 is sent before the timestamp punctuation tp_1. The modification punctuation is never buffered by any operator, so it cannot "travel" slower than the timestamp punctuation, meaning mp_1 must arrive before tp_1 on all components. This means that the same set of window instances will be active, which is exactly the set of window instances which is to be modified. The window instances which are processing and completed remains unmodified.

7. EXPERIMENTS

This paper combines many diverse techniques into one complete framework. A simple job is used for the experiments which are not sensitive to the size of the job, while

large jobs with up to 150 operators, are used for the rest of the experiments.

Simple Job. The job consists of one compute operator and one consolidate operator, which together conduct aggregation over multiple windows. The compute operator maintains partial results over one minute, while the consolidate operator combines these into multiple complete results. The job can be executed as one stateful rigid component (combined) or as two components (split), where the first is stateful flexible and the second is stateless rigid.

Large Job. The job is generated randomly in two steps: (1) assigning partitioning attributes and (2) defining communication between operators. To assign partitioning attributes, an operator simply gets a random subset of 30 partition attributes, whose size is initially half of the partition attributes. The size of the subset is changed by a random value for each new operator and the actual partition attributes chosen with the given size, is always completely random. To define the communication between operators, we first randomly choose the number of levels in the topology, i.e. the longest path in the DAG, as a random value between three and 0.75· #operators. The operators are then divided into the needed levels, from the first to the last operator. Each operator sends X tuples to operators in the immediate downstream level. The communication is calculated by maintaining X and calculating the tuples to the first operator as a random between 0 and X, multiplied by a random between 0 and 1. This gives the amount of tuples, Y, to send to the first operator and X is updated to be $X = X - Y$, and the calculation is done similarly for the next operator in the downstream level.

Fast Job. The job consists of four consecutive compute operators, each doing aggregation over window with incremental intervals (1 minute, 5 minutes, 10 minutes and 20 minutes). Tuples are eagerly sent along from the first operator to the last operator. This job is used to show the benefits of co-partitioning in section 7.7.

Setting. All experiments are executed on Amazon EC2. One instance of type $m1.medium$ is used to execute Apache Zookeeper [15] and the Nimbus daemon (Apache Storm master). Six instances of type $m1.medium$ are used to process the job. Lastly, one $m3.xlarge$ instance is used to produce inputs. We employ a more performant instance to produce inputs, to ensure it can produce tuples as fast as the cluster can process them.

Input. We use synthetic data, which is initially distributed uniformly over a given keyspace. Using synthetic data allows us to change size of state, keyspace and more, which is useful when experimenting with specific features.

7.1 Consolidation Time

We first investigate how much the consolidation affects the throughput, when varying the number of tuples to consolidate, the number of overlapping windows and whether the *simple job* is executed as split or combined.

Setting. The experiment is executed under multiple configurations, the first produces complete results over two windows (each 5 and 10 minutes) and the second produces complete results over six windows (each 5, 10, 15, 20, 25 and 30 minutes). In order to calculate the throughput reliably, we ensure the input rate is a bit higher than the maximum processing capacity of the system, which for this experiment is 40000 tuples / second. We calculate throughput, based on how much time it takes to completely process 18 million tuples.

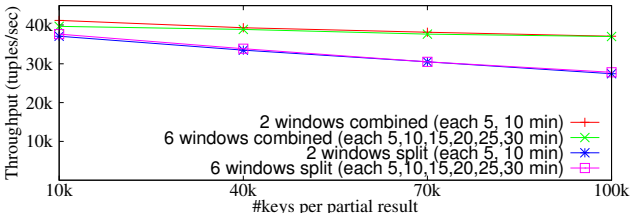

Figure 9: Consolidation Overhead

Results. Figure 9 shows the overhead of consolidation increases slowly and gradually with the number of tuples to combine (keys per partial result). The result shows that the overhead of consolidation is low, since doubling the number of tuples to consolidate only affects the throughput of the job very little (for the combined component). The overhead of consolidation affects the split component more, as the overhead of serialization and deserialization becomes larger. The number of overlapping windows has little impact on the throughput. It can be understood by considering that when using more windows, the consolidation frequency is actually lowered, which means the overhead is also lowered. This is a very common pattern. The figure shows the same trend for executing a split component, where the throughput is simply lower, due to the overhead of executing a split component under this configuration. Remember that our framework will choose the most performant execution technique at runtime, based on the cost-model defined in section 5.1.1.

7.2 Migration Latency

In this experiment, we examine how the latency (time processing is paused) resulting from migrating a workload between nodes, depends on the size of state.

Setting. The *simple job* is split, and executed with a suitable input rate to ensure the nodes are averagely loaded around 70%. The job is executed for four minutes to allow the Java Virtual Machine to do optimizations, before migrating a random key group between two nodes. The experiment is executed multiple times and both instant and disruptive migration are examined.

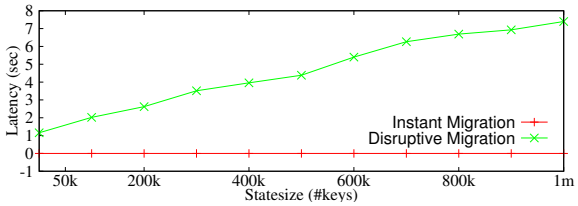

Figure 10: Migration Latency

Result. Figure 10 shows one line for instant migration and one for disruptive migration. Remember that instant migration is possible for all adaptations (except combine), while disruptive migration is the standard in many other frameworks and systems. Instant migration incurs negligible processing latency, because serialization only needs to be applied to the partial results, after the state has been built, for which reason it can be done in a thread which is separate

to the low-latency execution. For this experiment, disruptive migration was done with direct migration as described in section 3. The larger the size of state is, the more efficient it becomes to perform instant migration.

7.3 Split & Combine

We now investigate the overhead of applying split and combine, in terms of latency and completion time, which are calculated on the basis of the previous experiment (section 7.2). Split can always be done using instant migration, which means it finishes very fast with negligible processing latency. Combine can be done using disruptive migration or smooth migration. Smooth migration does not incur any latency, but has the maximum completion time of the longest window processed by the component.

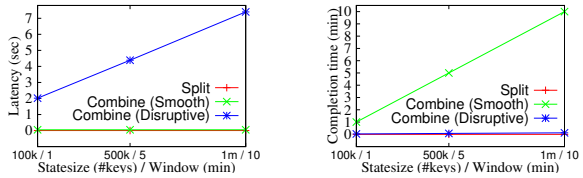

Figure 11: Split & Combine Overhead

Result. Figure 11 shows that split can always be done very efficiently. Combine with smooth migration incurs no latency, but can result in a long completion time. Combine with disruptive migration incurs latency relative to the statesize to migrate. The most efficient combine action should be chosen at runtime, using a suitable cost model [21].

7.4 Scaling a Flexible Stateful Component

In this experiment, we consider the performance overhead of applying instant migration and disruptive migration, on a flexible stateful component. The overhead is measured as the extra time needed to complete the processing.

Setting. In order to get a flexible stateful component, the *simple job* is split. The compute produces a partial result per minute and the consolidate produces a complete result every five minutes. The initialization phase is 20 minutes, after which we migrate between 0 and 100% of the key groups of the compute operator using either instant or disruptive migration.

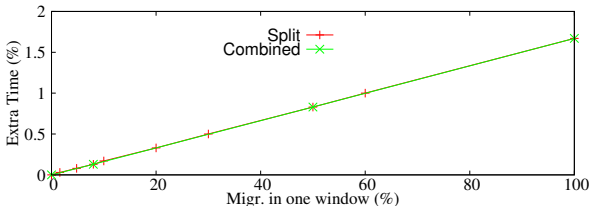

Figure 12: Overhead

Result. Figure 12 shows the percentual increase in completion time, from migrating x% state. The overhead is similar for both approaches. To see why, consider that using disruptive migration to move one key group incurs the overhead of serializing and deserializing the key group. If employing instant migration to migrate one key group instead, we get extra partial data corresponding to exactly one key group to also serialize and deserialize (at a later time). The benefit of the split component is that it can apply instant migration, where the combined component must apply disruptive migration, i.e. the split component benefits from low state migration latency without incurring any performance overhead.

7.5 Instant vs. Disruptive Migration

In this experiment we compare instant migration with disruptive migration, and show that for some inputs, instant migration is needed to obtain low processing latency. The *simple job* is used, since the result of this experiment is independent on the size of the job.

Setting. The first five minutes is the initialization phase. During the initialization, no adaptation is done and the input rate is fixed, such that each node is loaded around 70%. Afterwards, the input rate is changed as follows: A node is chosen and the input rate for the key groups allocated to the selected node, are gradually increased over the next three minutes, until 50% extra load has been added. This is implemented by skewing the key distribution at runtime. The node thus becomes overloaded and adaptation is needed to maintain a reasonable processing latency. After two more minutes, a new node is selected and the process is repeated. The job is executed under multiple configurations, (1) disruptive migration is applied each minute, (2) disruptive migration is applied each five minutes (3) and lastly instant migration is applied continuously.

The key space of the input is adjusted, such that each key group has around 50k keys. This is a very reasonable value, as it is large enough to introduce latency from the disruptive migrations, which heavily depend on the size of state to migrate.

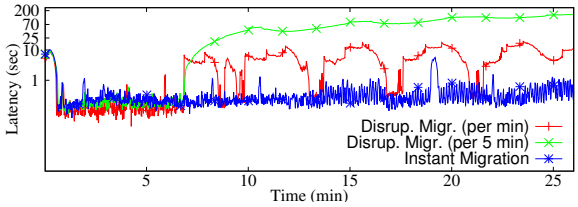

Figure 13: Latency

Result. The result is shown in figure 13. When disruptive migration is done every five minutes, it results in extensive processing latency because the slow migrations cannot keep up with the change in input and nodes become overloaded. This problem can only be handled by increasing the frequency of performing migrations.

Performing disruptive migration every minute results in better latency, because the migrations are now done frequently enough that the system is able to handle overloaded nodes faster. The processing latency in this case primarily comes from the actual state migration operations, which cannot be avoided.

Instant migration achieves a factor of ten improvement over the disruptive migration in terms of processing latency. Obtaining a low processing latency is important for multiple reasons. For instance, while processing fast (high input rate) input streams, a long pause can lead to either excessive memory usage or the worse, flushing tuples to disk if run-

ning out of memory. Therefore, incurring as little processing latency as possible is an important performance objective.

7.6 Fusion

In this experiment, we investigate how good our heuristics for automatic fusion are. Fusion depends heavily on the partitioning attributes and the operator configuration, for which reason we generate random jobs for 50, 100 and 150 operators with 30 partition attributes, as described in the beginning of this section. Each heuristic is executed on 500 random jobs and average results are presented.

Baseline. We define a baseline, which works by taking the first operator in the job and trying to merge downstream operators, by greedily fusing the operator whose intersection of partition attributes is the largest. This fusion operation continues until no more operators can be fused. Then it selects a random non-fused operator and retries the fusion operation.

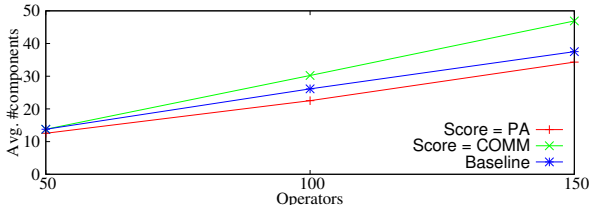

Figure 14: Avg. #Components

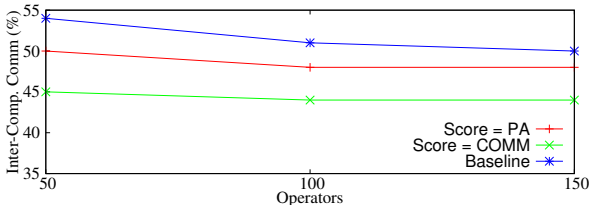

Figure 15: Avg. Inter-Component Communication

Result. Figure 14 shows the average number of components formed after applying our heuristics. The curve denoted as "score = PA" represents the score which tries to minimize the number of components, and the curve "score = COMM" represents the score which tries to minimize the communication cost directly. It is interesting to see that the approach which minimizes communication cost actually employs the largest number of components.

Figure 15 shows the inter-component communication as a percentage from 0 to 100. A lower level of inter-component communication means the better performance, since that minimizes the overhead associated with serializing and deserializing tuples.

Based on these results, the best approach is "score = COMM", since that gives the largest number of components (i.e. we expect it has good capability of parallelization) while obtaining the lowest inter-component communication.

7.7 Performance vs #Components

In this experiment, we consider how the throughput of the *simple job* varies with the number of components. The job is executed as either one (combined) or two (split) components. The throughput is calculated based on the time it takes to process 18 million tuples, calculated as the time from initializing, until the last tuple is processed by the last component.

Result. The result is that processing the job as one component takes 456 seconds and processing the job as two components takes 512 seconds. This is a difference of more than 10%, which shows that minimizing the number of components is very important for the throughput performance. The reason that the throughput is heavily dependent on this, is that all tuples must be serialized and deserialized for network transferring between components, which incurs a large cost.

Fast Job. In the next experiment we apply the automatic fusion algorithm to the *fast job*. The fusion algorithm is able to merge all operators into one component, but for this experiment we give the performance results with zero fused, two fused, three fused and four fused operators, as this can present the reader more insight into the importance of fusing.

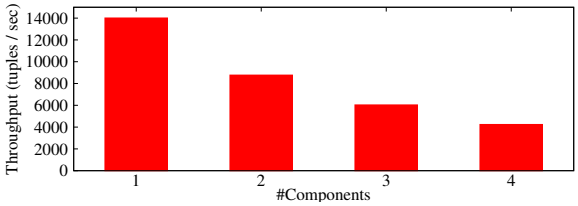

Figure 16: Completion Time

Result. As can be seen from the results (figure 16), the overhead associated with serializing and deserializing tuples is very significant. Our automatic fusion algorithm can improve the performance by more than a factor of three.

Notice that for this experiment, the calculation performed by each operator was a simple aggregation which can be done very efficiently. The more expensive the calculations done by the operators are, the less the impact of fusion will be, as the system overhead will no longer dominate the processing cost. According to our experience, most operator logic is quite simple and efficient to execute.

8. FUTURE WORK

Fusing operators into components can be optimized by allowing the fusion of only a subset of operator instances. This is useful, when only a subset of the operator instances are exhibiting extensive communication. Fusing operators which has little communication is actually not improving performance noteworthy, but instead potentially limiting the scalability of the solution.

The operator communication patterns can change at runtime, which means the system might need to fuse or unfuse operators at runtime. This can be handled by applying state migration techniques to ensure the operator instances to fuse are allocated to the same node. The astute reader will see that this problem is tightly coupled with load balancing.

9. CONCLUSION

In this paper, we present Enorm, a semantical layer over Storm with native support for window management. The framework not only simplifies the development of complicated window-based computations, but also enables efficient

sharing of computations on overlapping windows an novel runtime adaptations. All adaptations (except combine), can be done using Instant Migration, which exhibits both negligible completion time and negligible processing latency. Finally, we show that the improved semantics can also be used to transparently minimize the communication cost by grouping consistent operators into components. We propose an optimization algorithm and our experiments show that the optimization can improve the throughput of the tested job by a factor of three.

10. REFERENCES

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[2] D. Alves, P. Bizarro, and P. Marques. Flood: elastic streaming mapreduce. In *DEBS*, 2010.

[3] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive online scheduling in storm. In *DEBS*, 2013.

[4] M. Anis Uddin Nasir et al. The power of both choices: Practical load balancing for distributed stream processing engines. In *ICDE*, 2015.

[5] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, 2004.

[6] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD*, 2005.

[7] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. Scalable and low-latency data processing with stream mapreduce. In *CloudCom*, 2011.

[8] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, 2013.

[9] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 2008.

[11] L. Fischer and A. Bernstein. Workload scheduling in distributed stream processors using graph partitioning. In *Big Data*, 2015.

[12] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD*, 2008.

[13] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic scaling for data stream processing. IBM Research Report, RC25401 (WAT1308-061), 2013.

[14] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 2012.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIXATC*, 2010.

[16] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, 2005.

[17] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. Cola: Optimizing stream processing applications via graph partitioning. In *Middleware*, 2009.

[18] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, 2006.

[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *SIGMOD*, 2012.

[20] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow.*, 2012.

[21] K. G. S. Madsen and Y. Zhou. Dynamic resource management in a massively parallel stream processing engine. In *CIKM*, 2015.

[22] K. G. S. Madsen, Y. Zhou, and J. Cao. Integrative dynamic reconfiguration in a parallel stream processing engine. *arXiv:1602.03770*, 2016.

[23] K. G. S. Madsen, Y. Zhou, and L. Su. Grand challenge: Mapreduce-style processing of fast sensor data. In *DEBS*, 2013.

[24] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. *arXiv:1510.05714*, 2015.

[25] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, 2010.

[26] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell. R-storm: Resource-aware scheduling in storm. In *Middleware*, 2015.

[27] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an Elastic Stream Computing Platform for the Cloud. In *CLOUD*, 2011.

[28] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.

[29] L. Su and Y. Zhou. Tolerating correlated failures in massively parallel stream processing engines. In *ICDE*, 2016.

[30] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *SIGMOD '14*, 2014.

[31] J. Xu, Z. Chen, J. Tang, and S. Su. T-storm: traffic-aware online scheduling in storm. In *ICDCS*, 2014.

[32] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[33] Y. Zhou, K. Aberer, and K.-L. Tan. Toward massive query optimization in large-scale distributed stream systems. In *Middleware*. Springer, 2008.

[34] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *OTM*. Springer, 2006.