Aeolus: An Optimizer for Distributed Intra-Node-Parallel Streaming Systems

Matthias J. Sax[#12], Malu Castellanos[*2], Qiming Chen[*2], Meichun Hsu[*2]

[#]*Databases and Information Systems Group, Humboldt-Universität zu Berlin*
[*]*Hewlett-Packard Laboratories, Palo Alto (CA)*
[1]mjsax@informatik.hu-berlin.de [2]{firstname.lastname}@hp.com

***Abstract*—Aeolus is a prototype implementation of a topology optimizer on top of the distributed streaming system Storm. Aeolus extends Storm with a batching layer which can increase the topology's throughput by more than one order of magnitude. Furthermore, Aeolus implements an optimization algorithm that computes the optimal batch size and degree of parallelism for each node in the topology automatically. Even if Aeolus is built on top of Storm, the developed concepts are not limited to Storm and can be applied to any distributed intra-node-parallel streaming system. We propose to demo Aeolus using an interactive Web UI. One part of the Web UI is a topology builder allowing the user to interact with the system. Topologies can be created from scratch and their structure and/or parameters can be modified. Furthermore, the user is able to observe the impact of the changes on the optimization decisions and runtime behavior. Additionally, the Web UI gives a deep insight in the optimization process by visualizing it. The user can interactively step through the optimization process while the UI shows the optimizer's state, computations, and decisions. The Web UI is also able to monitor the execution of a non-optimized and optimized topology simultaneously showing the advantage of using Aeolus.**

I. Introduction

Processing large amounts of data has become increasingly important over the last years. The latest trend in "Data Intensive Computing" is the need for shorter processing times up to real-time analysis. Popular batching systems like Hadoop [1] do not meet the requirements for low latency processing. Nevertheless, the MapReduce programming model provides valuable properties for exploiting data parallel computation. Inspired by the MapReduce programming model and system architecture, new distributed streaming systems supporting intra-node parallelism have been developed to address the low latency requirements. Examples are Twitter's Storm [2], Yahoo!'s S4 [3], and Google's Percolator [4]. All three systems have in common that they execute dataflows in form of directed acyclic graphs (DAG), apply user-defined imperative code to data streams, and exploit data parallelism for the computation. However, they differ in tuple processing semantics (e. g., at-least-/at-most-once) and fault-tolerance guarantees.

Streaming systems are designed for low latency processing. Each output tuple of each processing node is sent to all consuming nodes immediately. Sending tuples individually may result in a big messaging overhead reducing the throughput of the system. In order to increase system throughput, *batching* techniques can be used [5]. However, it is not trivial to decide for which nodes batching is beneficial (i. e., increases the data output rate) and what the batching size should be. Furthermore, batching may influence what the best *degree of parallelism* (dop) for the consuming nodes is.

In this paper, we introduce Aeolus, an optimization algorithm that (1) decides for which nodes in the dataflow graph batching is beneficial and what the optimum batch sizes are and (2) computes the optimal degree of parallelism for each processing node. To the best of our knowledge there is no prior work on finding the optimal dop and/or batching size for a parallel stream dataflow program.

We implemented Aeolus on top of the open source system Storm [2]. Aeolus consists of a topology optimizer and a batching layer on top of Storm (Storm does not support batching natively). We integrated Aeolus with a Web UI in order to demonstrate how Aeolus works. Our demo system provides a topology builder that allows the user to interact with the system. New topologies can be designed and properties like dop, batching size, average processing time for individual tuples, fan-out (i. e., input/output ratio), and average output tuple size can be set for each processing node individually. Our demo system allows the user to submit topologies to the optimizer and observe the optimization process. We visualize the optimizer's state, computations, and decisions enabling the user to follow the whole process step-by-step. The visualization of the optimization process can be paused at any step to obtain explanation of the optimizer's computations and decisions in detail. Furthermore, the user specified topology and the optimized topology can be submitted to a Storm cluster and the parallel execution of both is monitored simultaneously. For example, monitoring shows the number of processed tuples over time, average tuple output rates for individual processing nodes, as well as network utilization for individual connections. Monitoring the execution of both topologies at the same time shows the runtime advantages of an optimized topology over that of a non-optimized. As the topology builder allows the user to define the dop and batch size for each node in the topology, we can also demonstrate that picking the right configuration is a non-trivial task.

II. Optimization Problem

Computing the optimal batch size and degree of parallelism (dop) for each node of a dataflow graph can be formulated as an optimization problem. Here, we give some definitions and terms to formulate our optimization problem precisely.

Furthermore, we introduce our optimization algorithm which calculates an optimal solution (a proof is future work).

Definition 1: A *dataflow* $D = (V, E)$, is a connected directed acyclic graph (DAG) consisting of a set of nodes V and a set of edges E. E contains tuples (v_i, v_j) where $v_i \in V$ and $v_j \in V$. Each node v in the graph has a label dop_v that is a natural number describing the degree of parallelism this node has for execution. During execution, each node v will be executed in dop_v many *tasks* in parallel.

Each node in the graph can emit tuples which are sent via the directed edges from the producing node to the consuming nodes. There are different connection patterns by which tuples are sent to the consumer tasks. The two most common patterns are "random" and "key-based". Without loss of generality, we will assume random pattern in the remainder of this paper.

Definition 2: If a task outputs tuples $t_1, \ldots, t_n$ while processing an input tuple t, we call $t_1, \ldots, t_n$ *child tuples* of t. Each input tuple of any node forms a *processing tree*. The processing tree $PT(t)$ of tuple t consists of t and all *recursively* emitted child tuples, i. e., all children of all child tuples and so on.

Conceptually, we assign two time stamps to each tuple t: *t.create* and *t.delete*. The create time stamp contains the time when the tuple is newly created and delete indicates the time when the tuple is 'deleted', i. e., completely processed. We can now define the *latency of a tuple*, which is the time it takes to process its processing tree.

Definition 3: Let $PT(t)$ be the processing tree of tuple t. The *latency* $l(t)$ of a tuple t is: $l(t) = max(t'.delete | t' \in PT(t)) - t.create$.

The objective of our optimization problem is to minimize the average latency for all incoming tuples while using minimum resources. Let I be the set of all tuples emitted by all $s \in S$, where S is the set of source nodes having no incoming edges ($S = \{s \in V | \nexists (v, s) \in E\}$). Our optimization problem is defined as:

$$(\min \ \text{avg}(l(t)|\forall t \in I)) \wedge (\forall n \in V \backslash S : \min dop(n)) \quad (1)$$

Optimization Algorithm

So far we have defined what the *latency* of a tuple is. The question to answer is how we can *calculate* it. Conceptually, we assign two more time stamps to each tuple t: *t.arrival* and *t.start*. The arrival time stamp indicates the time when the tuple arrives at the consumer (and is put in the consumer's input queue) and start is the time when the consumer starts to process the tuple. We define the *pure processing time* of a tuple within a node as the time needed by the node to process the tuple completely (t.delete − t.start). The *queuing time* is t.start − t.arrive and the *shipping time* is t.arrive − t.create. The *actual processing time* of t is shipping time + queuing time + pure processing time and is directly related to the latency of a tuple. In order to minimize the average latency of all input tuples we have to minimize the average of the actual processing time for each node in the dataflow.

We cannot influence the pure processing time for a single tuple, but we can influence the average shipping and queuing time. Increasing the dop is expected to reduce the average queuing time as the input latency augments as more tuples get to be processed in parallel. Introducing batching increases the shipping time for individual tuples, but reduces the average shipping time because the fixed message overhead is shared over all tuples in a batch evenly. However, batching increases the average queuing time because all tuples of a batch are sent to the same consumer task and are inserted into the consumer's input queue at once. We have to resolve this trade off. From a theoretical point of view, increasing the dop for each node will result in better performance in any case, but in reality each running task puts some overhead on a machine and we want to minimize this overhead. More precisely, for each node we want to find the minimum dop that minimizes the average of the actual processing time.

1) Minimum dataflow example: We illustrate how the optimal producer batch size and consumer dop can be calculated with the smallest (and simplest) possible dataflow consisting of a single source (with dop 1) and single consumer. The producer batch size has to be chosen such that the producer output rate is maximized. The consumer dop must be big enough to avoid overloading of the consumer.[1] Let r_i be the data arrival rate (in tuples per second) for the consumer and ppt be the consumer's pure processing time for a single tuple (in seconds). The optimal dop for the consumer is:

$$\min\{\text{dop}_c | \text{dop}_c > ppt \cdot r_i\} \quad (2)$$

The data arrival rate of the consumer is equal to the data output rate of the producer ($r_o(p) = r_i(c)$). The input/output latency l_i, l_o[2] is the time between the arrival/transmission of two tuples, i. e., it is the inverse of an input/output rate ($l = r^{-1}$). In the following we use both terms interchangeable picking the more appropriate to the context in each case. The data output rate of the producer depends on multiple factors: (1) the data input rate of the producer, (2) the pure processing time of the producer, and (3) the shipping time to the consumer. We point out that even a source has a data input rate because it must fetch data externally. Therefore, we don't need to distinguish different types of nodes. However, sources often have a fixed dop of one, and our algorithm will not calculate a dop for sources but use the user defined dop.

The shipping time can be calculated as $n + t \cdot s$, where n is the fixed messaging network overhead in seconds, t is the average output tuple size in bytes, and s is the messaging data size overhead in seconds per byte. Given our example topology we can calculate the source output rate using no batching as follows: $r_o = (\max\{l_i, ppt + n + t \cdot s\})^{-1}$. If $ppt + n + t \cdot s \leq l_i$, batching is not beneficial because the time for processing and transmitting an output tuple is smaller than the time until the

[1] A node is overloaded if the pure processing time is too big compared to the input arrival rate, i. e., the input queue will grow over time as tuples arrive faster than they can be processed.

[2] Not to be confused with the latency of a tuple.

next input tuple arrives (i. e., the source has to wait until it can fetch the next external input). However, if $ppt + n + t \cdot s > l_i$ the producer's output rate is limited by the pure processing and shipping time. In this case, reducing the average shipping time by batching increases the output rate. We can calculate the output rate with batching as:

$$r_o = \frac{b}{\max\{b \cdot l_i, n + b \cdot (ppt + t \cdot s)\}} \tag{3}$$

If we enable batching with batching size b, we can transmit b tuples in time $n + (b \cdot t \cdot s)$. The optimal batch size reduces the shipping time such that shipping plus pure processing time is smaller than l_i. Using a bigger batch size does not increase throughput anymore while increasing the latency. Therefore, we calculate the optimum batch size as $b \cdot l_i \geq n + b \cdot (ppt + t \cdot s)$:

$$\Leftrightarrow \min\left\{ b \,\middle|\, b \geq \frac{n}{l_i - ppt - (t \cdot s)} \right\} \tag{4}$$

We can reduce the average shipping with increasing batch size (assuming that the shipping time is the dominant term). As long as $n \gg t \cdot s$ and b is small, n dominates the cost. With increasing batch size b the savings decreases because the equation gets dominated by $b \cdot (t \cdot s)$. From a mathematical point of view increasing the batch size always decreases average actual processing time. In practice, the size of a batch is limited by the underlying network hardware and transfer protocol. In case of TCP/IP, the maximum useful batch size in 64KB, which is the maximum data size for a single TCP/IP package.

Algorithm 1 TopologyOptimizer

```
1:  P ← all source nodes
2:  while P is not empty do
3:      for all p ∈ P do
4:          if input latency is smaller than actual proc. time then
5:              b ← calculate batch size to reduce ship. time (Eq. 4)
6:              if b > B_max then b = B_max
7:                  increase dop of p to increase l_i
8:              end if
9:          end if
10:         calculate output rate r_o (Eq. 3)
11:     end for
12:     C ← all unprocessed nodes with known input rate r_i
13:     for all c ∈ C do
14:         dop_c ← calculate dop such that l_i > ppt (Eq. 2)
15:     end for
16:     P ← all c ∈ C with outgoing edges
17: end while
```

2) Generalization: The above example shows that the optimum dop and batch size for a consumer depend on some consumer properties, some producer properties (i. e., the producers output rate), and some hardware dependent constants (i. e., n and s). However, there is no dependency in the opposite direction. Therefore, we can start to optimize a topology beginning at the sources and propagating the decisions from the producers to the consumers throughout the whole topology. We propose our *TopologyOptimizer* algorithm (Algorithm 1) which starts by maximizing the average output rate for a source (`for`-loop, line 3-11), by computing

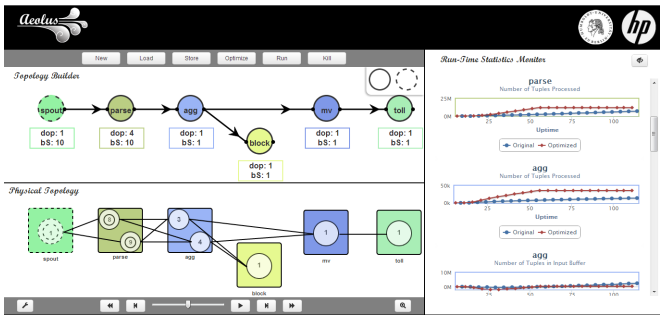

Fig. 1. Screenshot of the web user interface.

the optimal batch size.[3] The `if`-statement (line 4) checks if batching can increase the output rate and calculates the optimal batch size accordingly (line 5-8).[4] In the next step the output rate r_o—considering the dop of the producer—is computed (line 10). After all producers' batch sizes are optimized, TopologyOptimizer computes the needed dop for the consumers (line 12). Consumers are those nodes for which all producers are already optimized, i. e., the output rate of all producers is known. In the `for`-loop in line 13-15, the optimal dop is calculated for each consumer. Before the next iteration of the algorithm starts (`while`-loop, line 2), all optimized consumers of the current iteration are set to be producers for the next iteration (line 16). TopoloyOptimizer terminates, if no producers are left. This condition will be met eventually, as the dataflow is a connected DAG and the algorithm traverses it following the directed edges beginning at the sources. Because each node in the dataflow is optimized once, TopologyOptimizer has linear complexity making it applicable to larges topologies.

III. Demonstration Setup

We implemented our optimizer on top of Twitter's Storm (version 0.7.0) including a Web User Interface. The web UI can be used to create Storm topologies which can be stored and load (see TopologyBuilder in the left upper part in Figure 1). We provide a few example topologies including a topology from the Linear Road Benchmark [6] and a sentiment analysis topology over a stream of tweets. We also provide some predefined spout and bolt implementation that can be used to build topologies from scratch. The predefined spouts/bolts do not perform any useful work but simulate pure processing time by sleeping. The user of the TopologyBuilder UI can parameterize each spout/bolt in the topology individually by providing values for simulated processing time or average output tuple size. We provide example topologies that can be modified by the user in order to "play" with the system. The user can also define the dop and batch size for each node. This feature can be used to show that defining the best parameters for both is a non-trivial task.

[3] As mentioned above, we do not optimize the dop for source nodes but take the user defined dop.

[4] The batch size is limited by B_{max} which is the maximum TCP/IP package size of 64K.

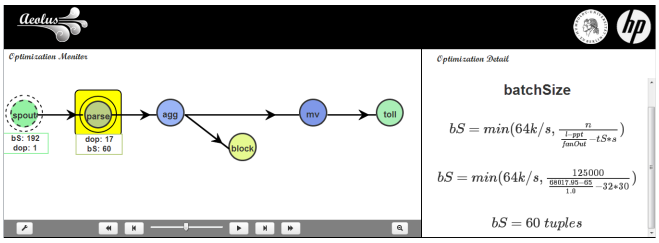

Fig. 2. Step-by-Step visualization of the optimization process.

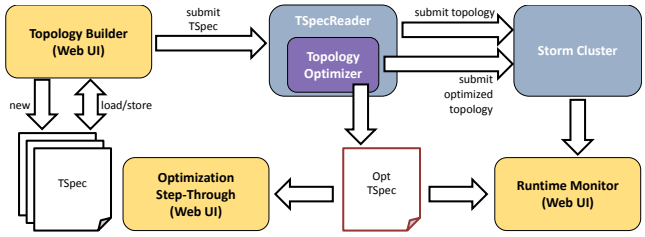

Fig. 3. Demo system architecture.

A Topology can be submitted to the optimizer and the UI shows how the optimizer steps through the topology and optimizes node by node (Figure 2). In 'play' mode all decisions made by the optimizer are visualized including its state (i. e., what node is processed right now, what are the already computer values of previously processed nodes) and the used formulas. The optimization can be stopped at any point allowing for a detailed explanation and discussion about the optimizers decisions. Figure 2 shows one optimization step calculating the batch size of a node. The currently processed node is highlighted with an orange background box and the used formulas instantiated with the actual values for the variables are shown. The nodes left (upstream) to the highlighted one are already optimized and the computed values for dop and batch size are displayed. The buttons on the bottom can be used to step forward and backward in the optimization process manually.

Furthermore, topologies can be submitted to a Storm cluster and the Web UI offers a live monitoring component (see right hand side in Figure 1) including statistical data about the execution (e. g., number of emitted tuples, average tuple size, or current network utilization). Submitting a topology always includes the execution of the non-optimized and optimized topology in parallel. Both topologies get monitored at the same time and corresponding statistics are displayed in a single chart. Therefore, the execution of both topologies can be compared with each other easily showing the advantages of using Aeolus. The UI also shows the *physical* topology (i. e., all parallel tasks and all network connections) in the left lower window during execution. The physical topology view displays more details about the execution like average batch buffer utilization and network load for individual connection. The physical topology can be displayed complete or in a more compact visualization (the compact visualization is useful if the dops are high resulting in too many details for the fully physical view).

IV. Implementation

We had to implement batching on top of Storm because Storm does not support batching techniques.[5] We implemented batching as wrapper classes where batching is transparent to Storm and to the user code. A batching wrapper gets an instance of an original spout or bolt as input and replaces the wrapped node in the topology. If batching is used for a node, all consumers must be wrapped with debatcher to reverse the batching operation before the actual user code in called.

The demo system architecture is shown in Figure 3. The Web UI components are shown in yellow boxes. TopologyBuilder at the top left lets the user design new topologies and save them as a file in our `tspec`-format. It can also load topologies including hand written topologies. TopologyBuilder calls `TSpecReader` which is able to process a topology in `tspec`-format and submits it either to the optimizer or to the Storm cluster for execution (or both, i. e., first to the optimizer and then the optimized topology to Storm). Our optimizer writes an optimization log (`TSpecOpt`) that is used by the UI to visualize the optimization process. We extended Storm to produce statistical information about the execution like output tuple counts. Our UI runtime monitor uses this information together with the optimization log in order to display the execution dashboards as well as the visualization of the physical topology including runtime statistics.

V. Conclusion and Future Work

In this paper, we introduced a novel optimization algorithm that computes the optimal values for batch size and degree of parallelism for each node in the dataflow. We implemented a prototype of our optimizer on top of the open source system Storm. Our experiments show, that batching allows for a speed up of one order of magnitude compared to tuple-by-tuple processing. Our optimizer is able to compute the best configuration for a topology making manual (and time consuming) tuning unnecessary. As future work we plan to apply our optimization adaptively on running topologies in order to deal with burst input rates.

Acknowledgment: We like to thank Carlos Ceja and Lupita Paz for helping us build some components of the demo.

[5]Note that Storm supports "Transactional Topologies" and uses the term *batch* in this context. However, *batches* in Transactional Topologies are *logical* and not *physical* like in our case. All tuples of a logical batch are sent individually over the network and can be processed by different consumer tasks.

References

[1] The Apache Software Foundation, "Hadoop," http://hadoop.apache.org/.
[2] N. Marz, "Storm: Distributed and fault-tolerant realtime computation," http://storm-project.net/.
[3] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *ICDM Workshops*, 2010, pp. 170–177.
[4] D. Peng and F. Dabek, "Large-scale Incremental Processing Using Distributed Transactions and Notifications," in *Proc. of the 9th USENIX Symp. on Operating Systems Design and Implementation*, 2010.
[5] B. Lohrmann, D. Warneke, and O. Kao, "Massively-parallel stream processing under QoS constraints with Nephele," in *Proc. of the 21st Int. Symp. on High-Performance Parallel and Distributed Computing*, ser. HPDC '12, 2012.
[6] "The Linear Road Website," http://pages.cs.brandeis.edu/~linearroad/.