Performance Optimization 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*

[1] `mjsax@informatik.hu-berlin.de`

[*] *Hewlett-Packard Laboratories, Palo Alto (CA)*

[2] `{firstname.lastname}@hp.com`

***Abstract*— The performance of intra-node parallel dataflow programs in the context of streaming systems depends mainly on two parameters: the degree of parallelism for each node of the dataflow program as well as the batching size for each node. In the state-of-the-art systems the user has to specify those values manually. Manual tuning of both parameters is necessary in order to get good performance. However, this process is difficult and time consuming—even for experts. In this paper we introduce and optimization algorithm that optimizes both parameters automatically. We define a novel cost model for intra-node parallel dataflow programs with user-defined functions. Furthermore, we introduce different batching schemes to reduce the number of output buffers, i. e., main memory consumption. We implemented our approach on top of the open source system Storm and ran experiments with different workloads. Our results show a throughput improvement of more than one order of magnitude while the optimization time is less than a second.**

I. Introduction

"Big Data" was recently characterized by Stronebraker by the 3 Vs: Volume, Velocity, and Variety. One requirement of velocity is low latency processing. MapReduce [1] systems which are very popular in the big data domain are a good solution to tackle high data volumes. However, being batch oriented they do not provide low latency. Inspired by the MapReduce programming model—which provides valuable properties for exploiting data parallel computation—, new distributed streaming systems supporting intra-node parallelism have been developed. Examples of intra-node parallel streaming systems are Twitter's Storm [2], Yahoo!'s S4 [3], and Walmart's Muppet [4]. All three systems have in common that they execute dataflows in form of directed acyclic graphs (DAG), apply user-defined imperative code (UDFs) 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.

The current parallel streaming systems focus on low latency processing. Each output tuple of each processing node is sent to all consuming nodes immediately. Moreover, those systems also need to deal with high data volumes at high data rates. Sending tuples individually may result in a huge messaging overhead reducing the throughput of the system. In order to increase system throughput, *batching* techniques can be used [5]. However, increasing the throughput by batching might make it necessary to increase the degree of parallelism (dop) of the consuming node because a higher dop decreases the input rate (increased by batching) of each parallel instance. Hence, batch size and dop are inter-independent and it is not trivial to decide for each node in the dataflow what the optimum batching size and dop is.

In this paper we address the above problem and make the following contributions:

1) We introduce different batching techniques for data-parallel streaming systems.
2) We formally define a cost model for UDF centric dataflow programs.
3) We develop an optimization algorithm that computes the optimal degree of parallelism and batch size for each node in a parallel dataflow program.
4) We implemented a batching layer and optimization algorithm on top of Storm and evaluate our approach with workloads of widely different characteristics.
5) The approach taken to implement the batching layer is non-intrusive and does not require any change neither to Storm nor to the user program.

We implemented our optimizer [6] on top of the open source system Storm, a distributed intra-node parallel streaming system. Storm executes so-called *topologies* that are dataflow programs specified as directed acyclic graphs. The tuples are sent over the directed edges from node to node and each node in the dataflow graph executes an imperative user-define function similar to the MapReduce [1] programming model. While we build our system on top of Storm, the developed concepts are applicable to other intra-node parallel streaming systems as well. To the best of our knowledge there is no prior work on finding the optimal dop and batching size for a parallel stream dataflow program.

II. Batching Techniques for Data-Parallel Streaming Systems

Intra-node parallel streaming systems execute dataflow programs in a parallel manner, i. e., each node in the dataflow program is executed in multiple instances (on maybe different) machines. Within a node the actual processing happens by executing user-defined functions. Formally, we define the programming model as follows:

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.

We call the defined dataflow program D the *logical* dataflow. For execution, this logical dataflow in expanded into a *physical* dataflow by replacing each node v in the dataflow with dop_v many task-nodes $v^1, \ldots, v^{dop_v}$. The edges between two logical nodes (p and c) are replaces by a set of edges that form a bipartite edge pattern from all producer task-nodes to all consumer task-nodes, i. e., the directed edge (p, c) is replaces by the set of directed edges $(p^1, c^1), \ldots, (p^1, c^{dop_c})$, $(p^2, c^1), \ldots, (p^2, c^{dop_c})$, $\ldots, (p^{dop_c}, c^1), \ldots, (p^{dop_p}, c^{dop_c})$.

Each node in the graph can emit tuples which are sent via the directed edges from the producing node to the consuming nodes. In the physical dataflow program each producer task is connected with all tasks of the corresponding consumer node. There are different connection patterns by which tuples are sent to the consumer tasks. The two most common patterns are "random" and "key-based". A random pattern allows a producer task to send an output tuple to *any* consumer task (e. g., round robin scheduling could be used). Key-based pattern uses a hash-partitioning strategy based on some user-specified attributes of the tuples, the *key attributes*. Hence, all tuples having the same key attribute values are sent to the same consumer task (all producer tasks of a given node use the same hash function to meet this requirement). Because streaming systems are designed for low latency processing, each emitted tuple is sent from the producer to the consumer individually, i. e., as an independent TCP/IP message. Because it is common for tuples to be small in the number of bytes often this strategy raises a high overhead as sending TCP/IP messages is costly.

Batching is a well known technique [5] to increase the throughput of streaming systems. In non-data-parallel systems it is straight forward to design and implement output buffers for batching. In intra-node parallel streaming systems exploiting data parallelism a logical input stream is (hash-) partitioned into physical sub-streams. (If partitioning does not happens,i. e., the tuples are distributed randomly, a single output buffer is sufficient.) Therefore, it is necessary to use multiple distinct output buffers which increases the main memory consumption of the system. Moreover, if the logical producer has multiple logical consumers even more buffers might be necessary. In the following we describe our basic buffer scheme for data parallel sub-streams. We also discuss some details about implementing a batching layer in user space, i. e., how to wrap UDFs with a batching layer. This wrapping approach has the advantage of being non-intrusive and transparent to the streaming system and to the user code.

Additionally, we introduce novel batching schemes in order to reduce the number of buffers in the case of multiple logical consumers. Namely we introduce the *distinct batching scheme* that applies to multiple consumers that use different keys for partitioning. The distinct scheme has the advantage that it uses only a small number of output buffers. However, since it might not always be applicable (depending on the streaming system),

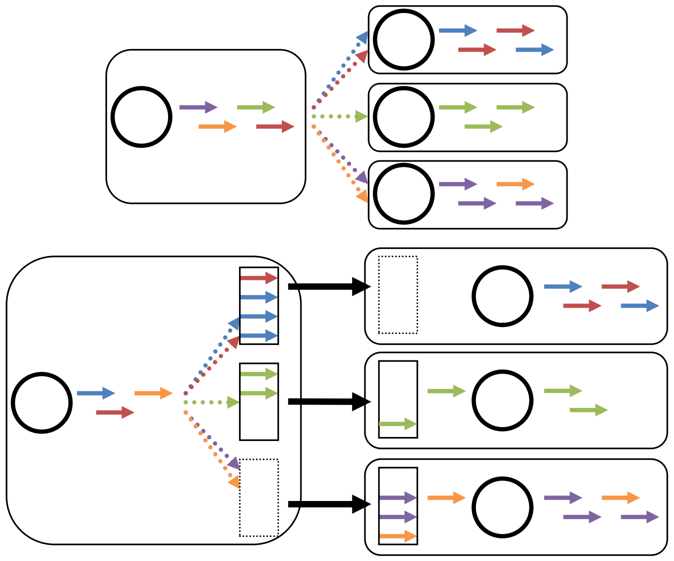
Fig. 1. A single producer task splits a stream into sub-streams (top w/o batching—bottom w/ batching).

we also introduce the *shared batching schema* that can be used in any case. However, shared batches incur in higher memory consumption, because more batching buffers are needed.

A. Batching for Key-Based Data Distribution

Intra-node parallel streaming systems exploit data parallelism and use (hash-)partitioning to split a logical stream into multiple physical streams according to a given key. This idea is inspired by the MapReduce programming model. Tuples having the same values in their key attributes [1] have to be processed "together"[2]. Because a single batch of tuples is sent to a single parallel consumer instance (a task), it must not contain tuples that belong to streams that are processed by different tasks. Hence, we need an output buffer for each consumer task, i. e., the degree of parallelism of the consumer determines the number of output buffers of the producer. This formula holds for a system like Storm in which a single task produces multiple sub-streams. S4 [3] for example, runs a task (called "Processing Element") for each sub-stream, i. e., the number of output buffers is determined by the number of distinct key values. Figure 1 illustrates a single producer task which splits a logical stream into physical sub-streams. Tuples are shown as arrows, and each color of the arrows corresponds to a different tuple key value. The Figure shows three consumer tasks, each processing one or two sub-streams. The bottom part of the Figure shows output buffers inserted into the producer (one for each consumer task) ensuring that tuples for a single consumer task are batched together.

[1]A key is not a unique identifier like in the relational data model.

[2]In MapReduce "together" means at once. This is feasible because MapReduce is a batch system. In the context of intra-node parallel streaming "together" means that a stream is split in parallel sub-streams—one sub-stream for each distinct key value.

Transparent Implementation: Our approach in implementing batching is non-intrusive, i. e., we do not alter the streaming system. Being non-intrusive has the advantage that a user does not need to install a modified version of the system. We built our library in a way such that it can be used without even altering the existing user code, in particular, the user-defined functions. Hence, our batching layer is transparent both to the system and to the user.

We implemented our batching on top of Storm as "regular" user-defined functions (from a Storm perspective). Additionally, our batching UDFs are parameterized with the original user functions. We implemented a wrapper to batch result tuples and a wrapper to de-batch incoming tuples. A batch is basically a fat tuple—we call it *batch tuple*. To Storm a batch looks like a regular (only very big) tuple. Because Storm performs some sanity-checks, we used a *horizontal layout* for our tuple. A simple batch tuple layout would be a concatenation of multiple tuples in a single tuple, either with a very large number of attributes or as a list of tuples. However, Storm would compare this tuple layout with the user specified tuple layout and would reject it, because the layout would not match. Our *horizontal layout* is inspired by the column-based table layout in the context of relational database systems. We split each tuple into its attributes and design the batch tuple as lists of attribute values. For example, two tuples $t_1 = \langle a_1, b_1 \rangle$ and $t_2 \langle a_2, b_2 \rangle$ are inserted into a batch tuple b like $\langle [a_1, a_2], [b_1, b_2] \rangle$. Therefore the batch tuple has the same layout (2 attributes) as the original tuples t_1 and t_2 and Storm accepts the tuples (Storm does not check the attributes types).

Additionally, we need to alter the default hash function used by Storm, i. e., the standard `.hashCode()` method that all Java classes have. Per default, Storm calls `.hashCode()` on the key attributes to determine the consumer task. Therefore we cannot use Java's standard list classes because this would change the hash value inconsistently. The Java list `.hashCode()` implementation uses all values in the list and produces a completely different hash value compared to the hash value of any single value in the list.

We implemented an `AttributeList` class that returns the hash value of the first attribute in the list. We could use any attribute because we know already that all of them have the same hash value—otherwise we would have inserted them in different buffers in the first place (see Subsection II-B for more details). Hence, the hash value of the batch tuple is the same as for each tuple in the batch and Storm computes the correct consumer task.

B. Batching for Multiple Consumers

In the case of multiple logical consumers we distinguish two main cases: (1) all logical consumers use the same key attributes and have the same degree of parallelism; (2) the logical consumers use different key attributes or have different degree of parallelism. Case 1 is the simple case. Because all consumers use the same keys and have the same dop, we can use the same output buffers for all consumers. Each buffer is now sent to one task of each logical consumer. The batching scheme is the same as for a single consumer. For case 2 we distinguish the following two schemes: *distinct batches* and *shared batches*. We need to use these advance batching schemes to avoid error prone routing.The top part of Figure 2 illustrates this situation. The producer has two logical consumers (white and gray background) with three and two parallel instances respectively. The upper consumer uses the key indicated by different colors (blue and red for the first instance, green for the second instance, and yellow and purple for the third instance). The lower consumer uses a different key indicated by the shape of the arrows (circle and square for

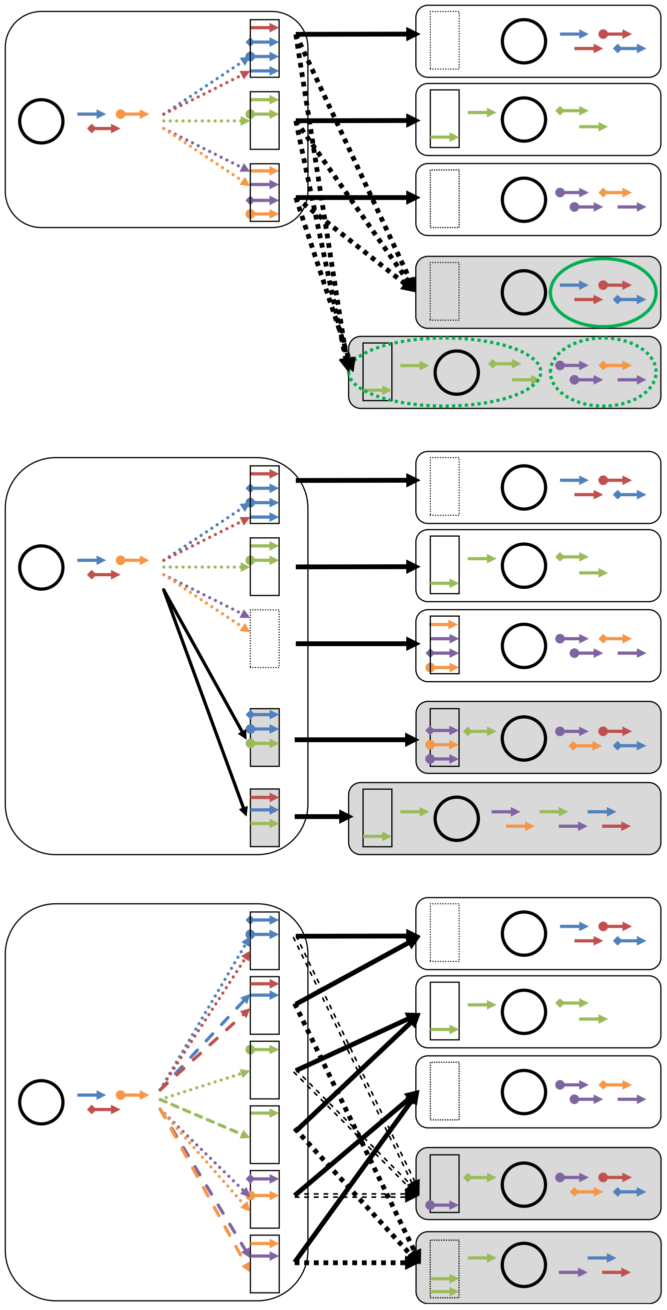
Fig. 2. Batching schemes for multiple logical consumers (top: error case—middle: distinct batches—buttom: shared batches).

the first instance, plain for the second instance). The producer builds the batches considering the color key only (as in the case of a single logical consumer). Therefore, batches contain tuples with arbitrary shapes. The batch marked with the solid green ellipse is routed to instance one of the lower consumer because the first tuple of the batch has square shape (◆⟶). The two batches (two green dotted ellipses) transmitted to the second instance have a plain tuple (⟶) which is used for routing. Because, each batch may contain tuples with arbitrary key values (i. e., arbitrary shaped tuples) tuples of any key might be processed by any consumer instance. This violates the partitioning criteria that all tuples having the same key must be processed by a single instance. In the following, we explain two different routing schemes that solve this problem.

1) Distinct Batches for Different Consumers: The distinct batching scheme uses a separate buffer for each consumer task of each logical consumer (middle part of Figure 2). These batches are logically grouped in buffer pools—one pool for each logical consumer (white/gray buffers for white/gray consumers respectively). The distinct batching scheme can be used if multiple logical consumers use different key attributes. In this case, each output tuples is inserted into multiple output buffers—one for each logical consumer, i. e., each tuple is inserted into one buffer of each buffer pool. For each buffer pool, a hash-function is used that considers the key attributes defined by the corresponding logical consumer of this buffer pool (tuples are hashed on colors for white buffers and hashed on shape for gray buffers). Hence, each tuple is inserted into a buffer of a pool, independently to all other pools.

The problem with implementing this scheme as a UDF wrapper is the following. Because the streaming system is not aware of the buffers, it sends all buffers to all logical consumers by default. Therefore, the system sends a buffer which belongs to a single consumer to a task for each consumer. Thus, the distinct batches scheme can be used only if this default system strategy can be modified by the user. The default broadcast strategy (to all logical consumers) must be replaced by specifying a single logical consumer for each batch (as shown in the middle of Figure 2). For the case in which the default replication strategy cannot be modified, we developed the *shared batches scheme* described next.

2) Shared Batches for Different Consumers: The idea of shared batching scheme is to prepare the output buffers in a way, such that the system can replicate each tuple to all logical consumers without violating the intended partitioning (bottom part of Figure 2). In order to achieve this goal, a quadratic number of output buffers is needed. Assume that logical consumer one has a dop of 3 while logical consumer two has a dop of 2. Shared batches uses $3 \cdot 2 = 6$ buffers which are conceptually ordered as a matrix with 3 rows and 2 columns. The matrix represent all possible pairs of a consumer 1 instance and a consumer 2 instance which will share the tuples in one of the buffers. For each tuple a buffer is selected a follows: The key attributes of the first consumer is used to select the row and the key attributes of the second consumer is used to select the column. The tuple is inserted in the buffer corresponding to the computed position in the matrix. This buffer selection scheme ensures that inserted tuples obey both partitioning requirements. Therefore, each buffer can be sent to a single consumer instance for each logical consumer without violating the partitioning.[3]

III. Optimization Problem

The aim of our optimization is to find a configuration (values for degree of parallelism (dop) and batch size for each node in the dataflow) that maximizes the throughput [6]. Maximizing the throughput is the same as minimizing the *average* latency over all processed tuples. At the same time, we want to keep both parameters (dop and batch size) as low as possible in order not to waste resources (dop) and not to increase latency unnecessarily (batch size). Our basic cost model is the following. Conceptually we assign four timestamps to each tuple: *t.create*, *t.arrive*, *t.start*, and *t.delete*. t.create is the time at which a tuple is newly created by a producer. At time t.arrive the tuple arrives at the consumer and is inserted into its input queue. The difference of t.arrive − t.create is the *shipping time*. When the consumer starts to process a tuple, we assign t.start to the tuple. We define the *queuing time* as t.start − t.arrive. After a tuple is completely processed we assign timestamp t.delete to it. Hence, the time needed by the node to process the tuple completely is the *pure processing time* (t.delete − t.start). The *actual processing time* of t is shipping time + queuing time + pure processing time.

We define the optimization problem using the above defined timestamps as well as the following definitions:

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.

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

The latency we want to minimize is the sum of the actual processing time over all nodes in the dataflow (assuming no branches for simplicity). Hence, we can optimize each node almost independently. The actual processing time in a node is

[3]We described shared batches scheme for two logical consumers. However, the idea can be extended easily by adding more buffers and ordering all buffers logically in an n-dimensional space for n logical consumers.

the sum of pure processing time, shipping time, and queuing time. 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 rate for a single consumer instance is reduced because 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 a single consumer. In general we do not optimize the dop for data sources but leave this to the user. The main reason is that many sources cannot be parallelized anyway. Furthermore, the sources define the initial input rate that the remainder of the dataflow has to be able to process. An optimizer cannot decide how fast data should be pushed into the system. We point out that a source must fetch data externally and has therefore an output rate r_o in tuples per second (or a latency l_o, i. e., the time it takes to fetch the next external tuple; $r_o = l_o^{-1}$). Knowing the output rate of a producer (in this case a source) we can compute an optimal batch size as follows. First we need to consider the shipping time that depends on some hardware dependent parameters n (fixed messaging network cost) and s (data cost per byte) as well as the size t of the tuples in bytes. If $n+t\cdot s \leq l_o$, batching is not beneficial because the time to transmit the tuple is smaller than the time until the next external tuple is available. However, if $n+t\cdot s > l_o$ the source's output rate is limited by the shipping time. In this case, reducing the average shipping time by batching increases the output rate. We calculate the optimum batch size as $b \cdot l_i \geq n + b \cdot (ppt + t \cdot s)$:[4]

$$\Leftrightarrow \min \left\{ b \,\middle|\, b \geq \frac{n}{l_i - ppt - (t \cdot s)} \right\} \tag{2}$$

After computing the batch size we can compute the producers output rate:

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

The output rate of a producer is the input rate of its consumer ($r_o(p) = r_i(c)$). Knowing the pure processing time of the consumer we can compute the minimum dop needed to process the data stream at the rate that its tuples arrive:

$$\min\{\text{dop}_c | \text{dop}_c > ppt \cdot r_i\} \tag{4}$$

Algorithm 1 TopologyOptimizer

```
1: $P \leftarrow$ all source nodes
2: while $P$ is not empty do
3:     for all $p \in P$ do
4:         if input latency is smaller than act. proc. time then
5:             $b \leftarrow$ calc batch size to reduce ship. time (Eq. 2)
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:    $C \leftarrow$ all unprocessed nodes with known input rate $r_i$
13:    for all $c \in C$ do
14:        $\text{dop}_c \leftarrow$ calculate dop such that $l_i > ppt$ (Eq. 4)
15:    end for
16:    $P \leftarrow$ all $c \in C$ 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 (i. e., ppt), some producer properties (i. e., $r_o(p)$), 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 the optimal batch size.[5] The `if`-statement (line 4) checks if batching can increase the output rate and calculates the optimal batch size accordingly (line 5-8).[6] 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

[4]This formula is valid for sources and processing nodes and contains the pure processing time (ppt) that is zero for source. Because ppt is zero for sources input latency is equal to output latency ($l_i = l_o$).

[5]We do not optimize the dop for sources but take the user defined dop.

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

TABLE I

OPTIMIZED CONFIGURATION FOR LR, COMPUTED BY THE OPTIMIZER.

	spout	parse	agg	mv	toll	block
dop	1	20	7	1	1	1
bS	192	2048	1	1	1	1

applicable to larger topologies. We claim that our algorithms computes the optimal solution for the optimization problem. However, a proof is future work.

IV. EVALUATION

We evaluated our approach with two different Storm topologies. First, we used a modified dataflow from Linear Road Benchmark [7], [8]. Second, we used a sentiment analysis dataflow running on a stream of Tweets. We ran our experiments on a cluster with 12 machines each equipped with 2 Intel Xeon CPUs (each 2 cores with 2.8GHz) and 6GB of main memory. The operation system was Red Hat Linux 4.1.2-52 (64bit). We used Storm 0.7.0 and Java 1.6.0. We set up a single Zookeeper instance, a single Nimbus node, and 12 Supervisors—one on each node in the cluster and each with 8 workers running. The Nimbus node is the master accepting jobs (topologies) to be executed while Supervisors are slave nodes (each running in a JVM) executing tasks in worker threads. The nodes are connected via a 1Gb Ethernet.

In our experiments we ran each topology in four configurations: (a) *optimized*: optimize dop and batching together; (b) *not optimized*: dop=1 for each node in the dataflow and batching is disabled; (c) *no batching*: optimized dop and batching disabled; (d) *only batching*: optimized batching and dop=1 for each node in the dataflow. In the following we use Storm terminology: source nodes are called *spouts* and processing nodes are called *bolts*.

A. Linear Road

The modified Linear Road (LR) dataflow consists of one spout reading data from a file (600MB, 12M records). A record reports a position of a vehicle in a highway segment and consists of the following attributes: Type, Time, VID, Spd, XWay, Lane, Dir, Seg, and Pos. Type is always 0 identifying the record as a position report while Time is the timestamp of the record. VID is a vehicle id and Spd the speed of the vehicle. XWay (express way id), Lane (lane number), Dir (driving direction), Seg (segment number of the expressway), and Pos (absolute vehicle position coordinates at expressway).

The spout sends (using shuffle pattern) the data to a *parse* bolt that parses the record string and splits it into its attributes. The result is partitioned with grouping pattern (key $\langle xway, dir, seg\rangle$) and sent to an *agg* bolt computing the average speed over all vehicles for each segment. The *agg* bolt has two consumers, *mv* and *block*, both connected to *agg* with grouping pattern using key $\langle xway, dir\rangle$ and $\langle xway, dir, seg\rangle$ respectively. While *block* computes the traffic capacity, *mv* computes the average speed over a window of size 5. The output of *mv* is sent to *toll* bolt (grouping pattern using key

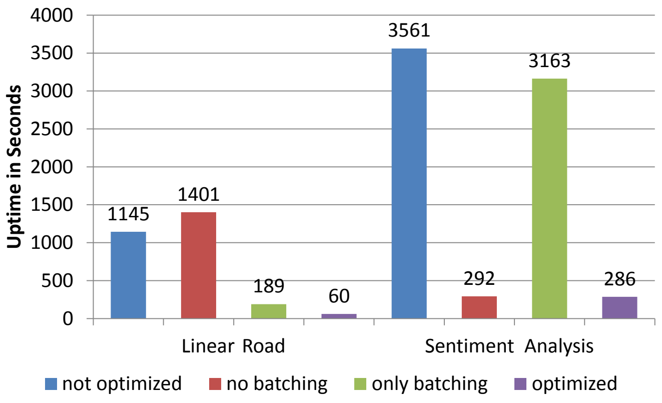

Fig. 3. Uptime of LR and SA topology with different optimizations.

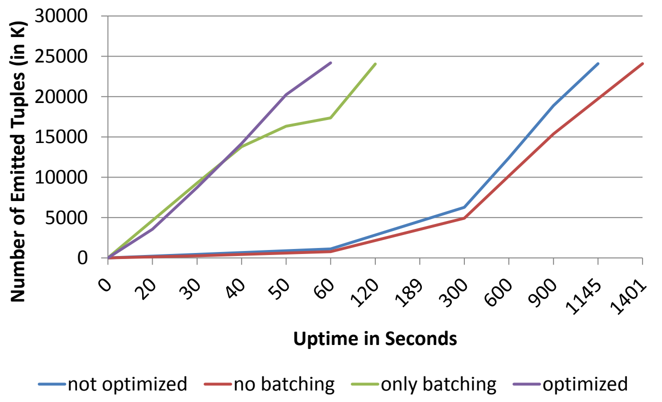

Fig. 4. Number of emitted tuples of LR with different optimizations.

⟨xway,dir,seg⟩) that computes the toll to be paid. Table I shows the configuration for degree of parallelism and batch size computed by the optimizer for each node in the dataflow.

Figure 3 shows the time needed to process all tuples from our fixed size data set for four configurations. The two cases without batching have about the same uptime of about 1250 seconds whereas the two cases with batching are 6 to 18 times faster. This result shows that batching gives the most speedup. Even for the 'batch only' case with dop of one for each node, speedup is significant. Using the optimizer for additional dop optimization gives a further improvement of a factor of 3. Considering the dop and batch size values in Table I we see that the *parse* bolt is the bottleneck of the dataflow because it got the highest dop of 20. At the same time, *parse* bolt has a high output rate resulting in a high batch size of 2048. The *agg* bolt also needs some parallelism to keep up with the output rate of *parse* bolt. However, *agg* aggregates the data resulting in a low output rate. Hence, batching is not needed. Furthermore, there is no need to parallelize any of the remaining nodes of the dataflow due to a low data rate.

Figure 4 shows the number of emitted tuples over all nodes over the uptime of the LR topology for our four configuration. Note, that the x-asis has no "proper" scale—it show various points in time when we gathered the number of emitted values. The figure shows that the two configurations with batching

TABLE II

OPTIMIZED CONFIGURATION FOR SA, COMPUTED BY THE OPTIMIZER.

	spout	filter	sentence	tok.	posTag	custN	cplx.P.	neg.	baseF.	sent.Word	att	norm.	stopWord	SA	html	res
dop	1[7]	53	2	3	18	2	1	2	2	2	2	2	2	2	3	3
bS	1	2	2	2	4	1	193	1	1	1	1	2	1	1	1	1
dop	1[8]	619	16	30	228	15	16	18	19	22	21	27	18	19	30	30
bS	606	374	270	284	250	250	193	164	140	124	111	99	99	72	46	366

(purple and green) perform about the same in the beginning. However, the non-optimized one (green) does not have the optimal dop and at about 40 seconds a bottleneck shows up. The number of emitted tuples does not increase with the same speed anymore. The reason seems to be that the *parse* bolt cannot buffer the tuples quickly enough any more. Because of the small dop, *parse* bolt cannot process tuples at their arrival rate and as the buffer size is limited, it seems to be full at 40 seconds. Hence, the spout has to slow down emitting tuples resulting in an overall performance loss. The other two configurations without batching have a worse performance compared to the two batching configurations.

We also measured the processing latency[9] for all the nodes in the dataflow. We report the latencies for the nodes involved in batching only, because the latencies for the other bolts did not change. Figure 5 shows the processing latency for *parse* and *agg* bolt. Latency is low for the case without batching and high for the cases with batching. Because Storm is not aware of batching, the processing latency of a batch tuple is measured (vs. measuring each tuple inside a batch). Because all tuples in the batch need to be processed before Storm considers the fat tuple to be processed, latency is much higher (pay attention that y-axis is in log scale). The same holds for *agg* bolt.

B. Sentiment Analysis

The Sentiment Analysis (SA) dataflow consists of one spout reading data from a file (140MB, 1M tweets), followed by a sequence of 15 bolts that perform the different steps of the analysis such as filtering the tweets by language, cleaning them, applying some natural language processing (e. g., tokenization, PoS tagging), discovering attributes, obtaining their associated sentiment, etc. The connection pattern is always shuffling. Because the spout reads the data from file, its output rate is very high. Optimization results in very high values for dop and batch size (two bottom lines of Table II). The sum over all dops is 1129. This high number of parallel tasks exceeds the capacity of our cluster. Thus, we reduced the output rate of the spout by disabling batching for it and optimized the remainder of the dataflow resulting in the configuration shown in the two top lines of Table II. Compared to the configuration computed for the LR topology, the result is more interesting. While the first bolt (*filter*) has a high dop of 53 (similar to *parse* bolt of LR), batching is determined to be only 2. The following two bolts (*sentence* and *tokenizer*) have a low dop and batch size. However, *pos Tagger* has a

[9]Storm defines process latency as “the time the bolt receives the tuple to the time it acks it”.

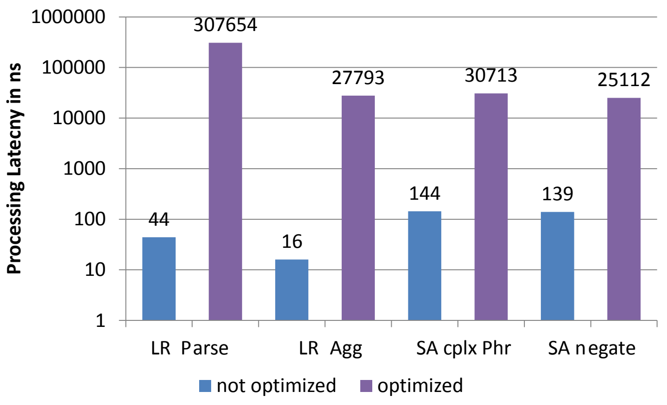

Fig. 5. Processing latency of selected bolts from LR and SA topology.

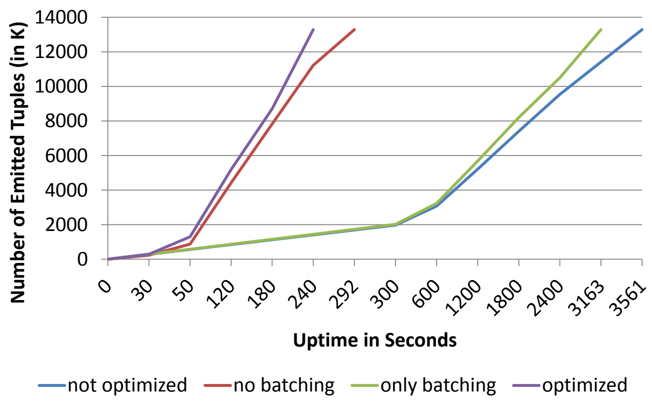

Fig. 6. Number of emitted tuples of SA with different optimizations.

high dop of 18. The reason is, that *pos Tagger* has a much higher processing time for a tuple compared to the other bolts. The remainder of the bolts have a low dop and batch size with one exception. *Complex Phrases* has a high batch size of 193. This is somewhat surprisingly because neither the processing time nor the size of the output tuples have an outstanding high value. This fact shows, that finding a good configuration manually is a very difficult task, even if the same meta data that the optimizer uses is available.

Figure 3 shows the time needed to process all tuples from our fixed size data set for our four configurations. We observe that batching does not give a significant speedup for SA dataflow (green bar w.r.t. blue one). However, parallelizing the dataflow results in a speedup of more than an order of magnitude. The fully optimized dataflow (purple) still gains 20% compared to the second best (red) in which batching is disabled. Figure 6 shows the number of emitted tuples over all nodes in the SA topology. The behavior is different to the LR case. During the first 30 seconds performance seems to the equally good over all configurations. After the first 30 seconds the optimized (purple) configuration has the best performance. We want to point out that the dop optimized configuration is the second best in contract to the LR case in which the two batching configurations are the top performers. We also measure the processing latency for all bolts with batches as input and/or output (Figure 5 show two selected bolts *complex Phrase* and *negate*). Similar to the LR result, processing latency increases for the configuration using batching. Especially, the high batch size of *complex Phrase* produces a significant increase in latency. Note that the high batch size is not only responsible for the latency increase of *complex Phrase* but also for the latency increase of *negate* (consumer of these big batches).

V. Related Work

Intra-node parallel streaming systems are a new class of distributed Streaming systems and our work applies to all of them. Storm [2], S4 [3], and Muppet [4] (now called MUPD8) are the first distributed intra-node parallel streaming system. In Storm the degree of parallelism for each node is static and must be set by the user for each node in the dataflow. Furthermore, Storm does not support batching of tuples resulting in low processing rates. S4 [3] is similar to Storm, however, it uses the notion of *Processing Elements* (PE) which are allocated dynamically. PEs are executed with many parallel instances each processing a parallel sub-stream (stream splitting happens randomly or key-based). PEs are executed by *Processing Nodes* (PN) and the number of PNs is static (set at S4 starting time). The user is not able to set a degree of parallelism and all PNs are used for each dataflow. Muppet [4] provides two functions map and update, each of which is executed parallelized. Muppet executes the map and update operators in workers, where each worker has a fixed number of threads. Each thread can execute any map/update instance and the assignment is dynamic. The number of workers is static (like the number of PNs in S4). Strom, S4, and Muppet also differ in processing guarantees (e. g., at-least-once or at-most-once), provide different level of abstractions to the user (spout/bolt, PE, map/update), and handle operator state differently. None of the systems uses batching to increase the throughput.

Lohrmann et al. [5] introduce dynamic batching within the Nephele system (which also supports intra-node parallel streaming) in order to meet QoS constraints. The aim of using dynamic batch sizes is to keep throughput as high as possible while meeting latency requirements. While this work is most similar to ours, there are main differences. First, optimizing the degree of parallelism is not part of their work. Second, while we focus on increasing the throughput, Lohrmann et al. optimize for QoS decreasing batch size in order to meet latency constraints which must be met by each tuple individually (we optimize average and not individual latency). COLA [9] is also an optimizer for stream dataflow programs. However, COLA optimizes the deployment process, i. e., the mapping from nodes in the dataflow to machines. COLA is built on SYSTEM S which does not support intra-node parallelism. Therefore, both approaches are orthogonal to each other.

VI. Conclusions 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 and a batching layer on top of the open source system Storm. Our experiments show, that batching allows for a speed up of more than one order of magnitude compared to tuple-by-tuple processing for some dataflow. Other dataflows gain most performance by setting an optimal dop for each node in the dataflow. In general, the combination of optimal batching and dop results in the best performance. 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. This extension must deal with the issue that operators can build up a state (e. g., sliding window). Changing the dop at runtime must ensure that operator state is preserved and "moved" between machines. Additionally, we want to enhance the optimization algorithm to handle resource limitations in the number of available workers in the cluster when optimal dop values exceed their number.

Acknowledgment

We like to thank Lupita Paz and Cornelio Inigo for their valuable help in the experiments.

References

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. of the 6th Conf. on Symp. on Operating Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 10–10.

[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 *Proc. of the 2010 IEEE Int. Conf. on Data Mining Workshops*, ser. ICDMW '10, 2010, pp. 170–177.

[4] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: MapReduce-style processing of fast data," *Proc. VLDB Endowment*, vol. 5, no. 12, pp. 1814–1825, Aug. 2012.

[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, pp. 271–282.

[6] M. Sax, M. Castellanos, Q. Chen, and M. Hsu, "Aeolus: An Optimizer for Distributed Intra-Node-Parallel Streaming Systems," *(Demo),* Will be presented at *Int. Conf. on Data Engineering,* 2013.

[7] "The Linear Road Benchmark Website," http://pages.cs.brandeis.edu/~linearroad/.

[8] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proc. of the Thirtieth Int. Conf. on Very large data bases - Volume 30*, ser. VLDB '04, 2004, pp. 480–491.

[9] 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 *Proc. of the 10th ACM/IFIP/USENIX Int. Conf. on Middleware*, ser. Middleware '09, 2009, pp. 16:1–16:20.