Building a Transparent Batching Layer for Storm

Matthias J. Sax, Malu Castellanos

HP Laboratories
HPL-2013-69

Keyword(s):
streaming data, distributed streaming system, batching, performance, optimization

Abstract:
Storm is a distributed intra-node-parallel stream processing system built for very low latency processing. One major drawback of Storm is its relatively low throughput. In order to increase Storm's throughput, we designed a batching layer for Storm that is able to improve Storm's throughput significantly. In order to get a high user acceptance, we did not modify Storm but build the batching layer "on top" of it. The layer is transparent to the Storm system as well as to the user code, i.e., the user-defined functions. Thus, already developed Storm programs (so-called topologies) can benefit from our batching layer without modification. In this document, we describe the design of the batching layer and provide inside into some implementation details.
Building a Transparent Batching Layer for Storm

HP Labs Technical Report HPL-2013-69

Matthias J. Sax
Databases and Information Systems Group
Humboldt-Universität zu Berlin, Germany
mjsax@informatik.hu-berlin.de
www.dbis.informatik.hu-berlin.de

Malu Castellanos
Hewlett-Packard Laboratories
Palo Alto (CA), USA
malu.castellanos@hp.com
www.hpl.hp.com

Abstract
Storm is a distributed intra-node-parallel stream processing system built for very low latency processing. One major drawback of Storm is its relatively low throughput. In order to increase Storm’s throughput, we designed a batching layer for Storm that is able to improve Storm’s throughput significantly. In order to get a high user acceptance, we did not modify Storm but build the batching layer “on top” of it. The layer is transparent to the Storm system as well as to the user code, i.e., the user-defined functions. Thus, already developed Storm programs (so-called topologies) can benefit from our batching layer without modification. In this document, we describe the design of the batching layer and provide inside into some implementation details.

1. Overview about Storm
Storm is a distributed intra-node-parallel stream processing system written in Java. The latest version of Storm is 0.9.1 and it is available open-source [1] under the Apache 2.0 License. Earlier versions of Storm are not part of Apache and are open-source under Eclipse Public License 1.0. In this section we describe Storm’s programming and execution model. We also give some details about the system model and include tiny code snippets when it is required for a better understanding. In Section 2, we are explaining how batching works in general and what difficulties occur using bathing within intra-node-parallel streaming systems. Furthermore, in Section 3 we are suggesting different batching schemes that solve the described problems. Section 4 covers the architecture of our batching layer and explains how we achieve a transparent implementation of top of Storm. Storm’s fault-tolerance mechanism is explained in Section 5. This section also explains how we ensure fault-tolerance within our batching layer. Finally, we conclude in Section 6. In the appendix, we show some examples that illustrate how our batching

1 matthias.sax@hp.com
layer works, i.e., what tuples are put into a batch and how those batches are handled in our layer including the interaction with the Storm system itself.

Programming Model
In order to use Storm, the user has to provide a dataflow program called topology. A topology is a directed graph consisting to two types of nodes: source nodes (with no incoming edges) called Spouts and processing nodes (with at least one incoming edge) called Bolts. For each node in the dataflow program a (imperative) user-defined function has to be provided. In order to specify a topology the user has to connect Spouts and Bolts in a connected graph. For each connection, a so-called connection pattern must be specified. We describe the available connection patterns and their purpose later in this section. An example dataflow program (or rather the structure of it) is shown in Figure 1. Each circle corresponds to a node. The two nodes in blue and red at the very left are Spouts (they do not have incoming edges) while the remaining four nodes are Bolts. The arrows represent the dataflow from node to node as directed edges. We also use the notion of a producer and a consumer node for each edge. The producer is the node where the tail of an edge is and the consumer is at an edge’s head. Notice that Spouts are always (and only) producers while Bolts are always consumers and can be producers. The two Bolts in the middle (green and purple) are producers and consumers at the same time (but for different edges). The orange and aqua colored nodes at the very right do not have outgoing edges; both are only consumers (also called drains).

If a Storm topology is executed, Spouts are requested to fetch data items from an external source and emit (i.e., output) those data items as tuples. A tuple consists of multiple attributes and each attribute has a name (the data type is not exposed to Storm). The emitted tuples from the Spouts are transferred along the edges to the consuming Bolt(s). If a Bolt (as the green Bolt in Figure 1) has multiple incoming edges, both data-streams are merged into a single stream before the data is given to the Bolt. Storm does not give any guarantees in which order the tuples are merged together. If a node has multiple outgoing edges (as the purple Bolt in Figure 1), the output is replicated to both consumers per default. If the data should not be replicated, the user has to tell the system explicitly to which predecessor nodes the data should be sent (we do not consider this case in this report). To this end, Storm uses the notion of logical stream. Each logical stream has a name and is logically separated from all other streams in the dataflow program. Per default there is only a single stream and we only consider this simplified case in this report. An extension of our technique to multiple streams is straightforward.

A Topology Implementation Perspective
Each Spout or Bolt is implemented by the user by implementing a Java interface or abstract class that is provided by Storm. The important methods for Spouts are open(...) and nextTuple() and for Bolts
are prepare(...) and execute(...). The open(...) and prepare(...) methods are called by Storm once when the topology is deployed. The method nextTuple() is called in an “infinite loop” requesting a Spout to produce a new output tuple. Each tuple emitted by a Spout is processed by the Bolts according to the specified topology structure. For Bolts, execute(...) is called for each input tuple. We want to point out that the user code can generate any number of output tuples per call of nextTuple() or execute(...). The user code can also preserve input tuples from previous calls, and use them later on to produce additional output tuples. To support this feature, both methods are declared as void and use a so-called OutputCollector for emitting output tuples. The OutputCollector is an object that is created by Storm at runtime and handed over to the Spout or Bolt in the open(...) and prepare(...) method respectively. The user has to preserve a reference to the OutputCollector object in an own member variable and use it within nextTuple() and execute(...). The OutputCollector provides a method emit(...) for returning an output tuple to Storm. (Basically, emit(...) is a back-call to Storm.). Example snippet for Spout and Bolt is shown below:

```java
public class UserSpout implements IRichSpout {
    private SpoutOutputCollector collector;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void nextTuple() {
        // use collector to emit output tuples
        this.collector.emit(...);
    }
}

public class UserBolt extends IRichBolt {
    private OutputCollector collector;

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
    }

    @Override
    public void execute(Tuple input) {
        // use collector to emit output tuples
        this.collector.emit(...);
    }
}
```

In order to connect Spouts and Bolts to a topology, the user uses the TopologyBuilder class. Each node in the dataflow program gets instantiated and “registered” at the TopologyBuilder. After registration the nodes are connected by specifying the connection pattern that should be used. The
listing below shows a tiny example in which a two node topology is built. Both nodes are connected with a *shuffleGrouping* connection pattern. We are going to explain the different connection patterns in the next section. For now we only point out that the consumer node determines the connection pattern, i.e., for a given consumer it is not possible to use any but only a dedicated pattern (that depends on the semantic of the consumer node).

```
TopologyBuilder builder = new TopologyBuilder();
// create source node
IRichSpout source = new UserSpout();
builder.setSpout(source);
// create processing node and use the source node as its producer
IRichBolt bolt = new UserBolt();
bolt.shuffleGrouping(source);
```

**Execution Model**

The Storm system has a shared-nothing architecture and runs in a compute cluster. It uses a master-worker-pattern. While the user only interacts with the master (called *nimbus*) the actual execution of a topology is performed by the workers. On each physical machine in the cluster a *supervisor* is running. The supervisor communicates with nimbus and distributes the assigned work to worker processes running on the same physical machine as the supervisor. Storm also uses the concept of *executors*. An executor is a thread within a worker and can process one or multiple *tasks* (see next paragraph). However, the executor concept is not important for this report and we do not explain it in more detail. We simply assume that each executor processes exactly one task, i.e., that the number of executors is equal to number of tasks. For this report, executors and tasks are not distinguished.

![Example Execution Graph](image)

*Figure 2: Example Execution Graph.*

Given a topology, Storm can execute multiple copies (called *tasks*) of each node in the dataflow program. To this end the user has to specify a *degree of parallelism (dop)* for each node in the dataflow program. Storm executes each node in its dop-many tasks (intra-node parallelism). We call the corresponding “expanded/physical” dataflow program execution graph. An example execution graph (that corresponds to the dataflow program from Figure 1) is shown in Figure 2. Each user-defined function must be written in a way, s.t., it works correctly with different dop values. This is necessary, because the dop is specified when a topology is created and not when the user code is written. Hence,
the user code must be independent from the dop. Note that each node in the dataflow program has its own dop and that all producer tasks are connected to all consumer tasks for each edge in the original dataflow program.

In our example topology, the blue producer Spout has a single consumer Bolt (the green node). In the expanded dataflow program, both blue Spout tasks are connected to all three consumer tasks (green). Because all three consumer tasks execute the same user-defined function, we want to partition the output data of the Spout over all three tasks. Note that we are at a task level now, in contrast to the node level. Hence, replicating data would not be beneficial because the same computation would be repeated multiple times. This principle is called \textit{data-parallelism} and is the reason why the user-defined code must be independent from the dop as explained above. The user has to specify how the data should be partitioned by specifying a so-called \textit{connection pattern} (as illustrated in the example code above), this pattern is specified when two nodes are connected while building a topology. The two most important patterns are \texttt{shuffleGrouping} and \texttt{fieldsGrouping}. Shuffle is a random distribution, i.e., Storm can send an output tuple to any consumer task. If \texttt{fieldsGrouping} is used, the user has to specify some tuple attributes (fields) – called \textit{keys} – to be used for hash-based partitioning. In this case, all tuples having the same values in the key fields are sent to the same task by Storm. It can happen that two different key-values have the same hash value but this does not violate the principle. Figure 3 shows a single producer task (circle) emitting tuples (solid arrows) with different key-attribute-values (indicated by colors—same color indicates equal key-attribute-values). The example shows a consumer with three tasks. The first task processes all blue and red tuples, the second one all green tuples, and the third task all purple and orange once.

Storm supports the additional connection patterns \texttt{allGrouping} and \texttt{globalGrouping}. The first one, \texttt{allGrouping}, is a replication pattern while \texttt{globalGrouping} sends all data to a single task. Additionally, there are two advance connection patterns \texttt{directGrouping} and \texttt{customGrouping} for expert users. We don’t cover any of those groupings in this report. For \texttt{allGrouping} and \texttt{globalGrouping} the same strategy

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Example for \texttt{fieldsGrouping} connection pattern.}
\end{figure}

\footnote{The only exceptions are Spouts/Bolts that are not parallelizable, i.e., they need to “see” all data to work correctly. Those Spouts/Bolts need to have a dop of one and this constraint must be respected when the topology is created. However, if a Spout/Bolt can have a dop greater than one, it should work correctly for any dop value.}
as for shuffleGrouping can be used within the batching layer. For directGrouping and customGrouping it is straightforward to extend the batching layer.
2. Batching for Data-Parallel Streaming Systems

So far, we briefly described Storm’s programming and execution model. In this section we give a brief overview about batching and how it can be integrated into a data-parallel stream processing system like Storm. Batching is a technique that groups data items together in order to transmit or process them at once for higher efficiency. Figure 4(a) shows a producer task (blue) sending output tuples (indicated as arrows) individually to a consumer task\(^3\) (green), while Figure 4(b) shows the usage of batches of size four.

![Figure 4](a) Individual tuple transfer. (b) Tuple transfer in batches.

In distributed streaming systems, the producer’s output data is usually sent to the consumer via a network connection. Sending a message over the network implies a certain overhead. Hence, sending each tuple individually results in more overhead than sending tuples in a batch. Each batch will be sent as one single network message; hence, a batch size of four reduces the overhead per tuple roughly by a factor of four. Having a single producer-consumer-pair, batching is simple to implement. In the case of intra-node-parallel operators, it is more complicated. In this case, the batching strategy depends on the used connection pattern. For shuffleGrouping, a simple batching schema as in the non-parallel case is sufficient. Any combination of tuples can be grouped together in a batch and any batch can be transferred to an arbitrary consumer task. In the case of fieldsGrouping such a simple batching scheme is not sufficient. Recall the example from Figure 3. In this case, putting red and green tuples into the same batch will result in wrong results because red and green tuples are supposed to be processed by different tasks. However, an entire batch must be transferred to a single task. Hence, our batching scheme creates an independent batching buffer for each consumer task. All tuples that will be processed by the same consumer are grouped together in the corresponding buffer. When a buffer is full, the batch of tuples is sent to the consumer task it belongs to. Figure 5 illustrates our method. Before the data is transferred (bold black arrow) from the producer task to a consumer task, the output tuples are partitioned (dotted colored arrows in the left box corresponding to the producer) according to their key values and inserted into different output buffers (of size four). Each output buffer is “connected” to a dedicated consumer task and will always be transferred to exactly this one. At the consumer side, the tuples are extracted from the batches and handed over to the user-defined code after each other. In our example we assume a single producer task. If the producer is executed in multiple tasks, we can employ our technique at each task independently to get the correct behavior.

In order to build the batches correctly, our method requires to know (or be able to access/call) the hash function which is used to compute the consumer ID by the underlying streaming system. Because Storm is an open source systems it was simple to find the corresponding code and to re-implement this hash-function in the batching layer. Putting the correct tuples together in a batch is only the first step. The

---

\(^3\) In the following we omit the word “task” from “consumer task” and “producer task” as this becomes obvious from the context.
The next question to answer is, how to design the batching layer, s.t. it is able to build those batches and how to hide the batches from the user-code and the Storm system.

*Figure 5: Batching for key-based data distribution.*
3. Batch Layout for Transparent Implementation

In order to provide a transparent implementation of batching on top of Storm it is necessary to design the layout of a batch carefully. The first step for transparent batching is the design of a batch-tuple. A batch-tuple models a batch as a regular tuple. The underlying Storm system must not know that the output tuple is not a regular data tuple but a batch tuple. In Storm, the user has to specify in advance what the number (and names, but not types) of attributes of an output tuple is. Therefore, the structure of the batch tuple must obey this specification (otherwise Storm will not accept the batch tuple). A simple concatenation of tuples (that we call “horizontal layout”) does not fulfill this requirement; the batch tuples will have as many attributes as the number of tuples (or [depending on the implementation], the number of tuples times the number of attributes per tuple). We call our batch layout “vertical layout” because we split each data tuple in its attributes and for each attribute we concatenate the attribute values of all tuples in the batch into a single value (e.g., we build a list of values for each attribute). The batch tuple contains these attribute batch lists as values (i.e., each list is one value). Therefore, the batch tuple has the same number of attributes as the original tuples and the streaming system will accept the batch tuple as if it was an original tuple.

Example: Given a batch size of four and four tuples $t_1 = (a_1, a_2, a_3)$, $t_2 = (b_1, b_2, b_3)$, $t_3 = (c_1, c_2, c_3)$, and $t_4 = (d_1, d_2, d_3)$ our vertical batching scheme builds a batch-tuple as: $b = ((a_1, b_1, c_1, d_1), (a_2, b_2, c_2, d_2), (a_3, b_3, c_3, d_3))$.

Figure 6: Building a batch tuple with three attributes and four input tuples.
Figure 6 shows the steps of building a batch tuple \( b \) and its (intermediate) layout having lists as values (using the example above). The lists are indicated as boxed within the batch-tuple \( b \).

Having the correct number of attributes for the batch tuple, we must ensure that the batch is transferred to the correct consumer task. Storm’s hashing algorithm uses the standard Java method `hashCode()`. This method is called for each tuple that is given to Storm via the emit method. If we use batches we want the hash value for the batch to be the same as for the tuples it contains, i.e., given a batch \( b \) containing tuples \( t_1, \ldots, t_n \), we want that \( H(b) = H(t_1) = H(t_2) = \ldots = H(t_n) \), where \( H \) is the hash function. We know already that all tuples within the batch do have the same hash value, i.e., that \( H(t_1) = H(t_2) = \ldots = H(t_n) \) — otherwise we would not have put them into the same batch in the first place.

In order to get the desired behavior, we do not use standard Java List-implementations but our own special list implementation for the batch (we call it `batch-list`). This is necessary for the following reason: usually, a hash value is calculated on the key-attributes-values of a producer output tuple in order to determine the corresponding consumer task. Because Storm does not know about batching, it would calculate the hash value for a batch tuple as if the key-attributes were single values (instead of lists of values). Therefore, the computed hash value could change and the batch would be sent to the wrong consumer (the standard Java list implementation considers all values in the list to compute a hash value for the entire list). Let us assume a batch size of 4 and tuples with 3 attributes (as in the example above). Let us further assume that attribute one and two are used for partitioning. Hence, \( H(t_i) = H(h(t_i.A_1), h(t_i.A_2)) \) and \( H(b) = H(h(<t_1.A_1, t_2.A_1, t_3.A_1, t_4.A_1>), h(<t_1.A_2, t_2.A_2, t_3.A_2, t_4.A_2>)) \). In general, we need to assume that \( h(t_i.A_1) \neq h(<t_1.A_1, t_2.A_1, t_3.A_1, t_4.A_1>) \) and therefore \( H(t_i) \neq H(b) \).

Our batch-list emulates the calculation of the (original) hash value, by picking an arbitrary value from each list. The hash value will be calculated using this value. As we know that without batching all tuples in a batch would have been routed to the same consumer task, we know that the hash value for each attribute in a batch-list is the same (which allows us to pick an arbitrary value). Figure 7 shows the
difference of hashing a regular tuple to hashing a batch tuple. As above we assume that attribute one and two are the key-attributes and that batch-list chooses the first value in each list to compute the hash value.

In the following we show code snippets for the batch-list implementation (BatchGroupColumn) that is a regular ArrayList that overwrites the hashCode() method to mimic the correct hash-value computation. A Batch is implemented as ArrayList with size of number-of-tuple-attributes using BatchGroupColumn to store the individual attribute values.

```java
class BatchGroupColumn extends ArrayList<Object> {
    public BatchGroupColumn(int batchSize, boolean monitoring) {
        super(batchSize);
    }
    public int hashCode() {
        return this.get(0).hashCode();
    }
}

class Batch extends ArrayList<BatchGroupColumn> {
    private static final long serialVersionUID = 2492749434454149161L;
    private int tupleSize;
    public Batch(int tupleSize, int batchSize) {
        super(tupleSize);
        this.tupleSize = tupleSize;
        // batch has column layout -> add a column for each attribute
        for(int i = 0; i < tupleSize; ++i) {
            this.add(new BatchGroupColumn(batchSize));
        }
    }
    public int length() {
        return this.get(0).size();
    }
    public void addTuple(List<Object> tuple) {
        // add each attribute of the tuple in its corresponding buffer column
        for(int i = 0; i < this.tupleSize; ++i) {
            this.get(i).add(tuple.get(i));
        }
    }
}
```

**Batching for Multiple Consumers using Different Key-Attributes**

So far we have described batching for a single logical producer-consumer-pair with field grouping connection. Multiple logical consumers (for one single producer) require an adaptation of our scheme if not all consumers are using the same key-attributes (different key-attributes are the common case). If
different-key attributes are used it could happen that two tuples having the same hash value for one consumer are having a different hash value for the other consumer. Therefore, we cannot put them into the same batch (being in the same batch would mean, that both tuples go to the same consumer task for each of the two logical consumers respectively). Figure 8 illustrates this case in which batches are routed in a wrong way. We indicate the keys of the two consumers with the color and shape of the arrows (colors for one consumer; shape for the other). Batches are built, considering the colors (one key) only. For consumer A using the colorkey everything is correct (top three consumer tasks with light grey background). The first task of them processes red+blue, the second one green, and the third one purple+orange. However, the second consumer B (bottom two consumer tasks with dark grey background) gets batches which are incorrect. We assume that the upper task is supposed to process all “shaped” tuples (squares+circles) while the lower one should process all tuple with a “plain” key. However, both consumer tasks retrieve tuples with all three different key-values (three different arrow-styles), which fails to meet the requirement that all tuples having the same key-values must be processed by a single task. The reason is that just the first tuple in the batch is considered for hashing. For the already processed batch marked with a green circle (solid line) the first tuple has a “square” key so the batch was sent to the upper consumer task (of the logical bottom consumer B). However, this batch also contains “plain tuples”. The same holds for the two batches marked with a green circle (dotted line). The first tuple in each of these two batches has a “plain” key and both are sent to the lower task of consumer B (however, both batches contain tuples with a “shaped” key belonging to the
upper task). We developed two different strategies to deal with this case correctly. Both strategies are explained in the following two sections.

**Distinct Batches for Different Consumers**

In our first strategy we use distinct batches for different consumers. In more detail, we use a distinct batch for each task for each consumer. In the example of Figure 8, we would use five batches because the first consumer has a dop of three and the second consumer has a dop of two. In order to do batching, each tuple is put into one batch for each consumer. Figure 9 illustrates the distinct batching scheme. Each tuple is put into two buffers according to the key-attribute-values of the corresponding consumer. The second consumer B (dark grey two task) now retrieves the tuples correctly grouped by the “shape”-key (i.e., the top task processes all “square”- and “circle”-keys, while the bottom task processes all “plain”-key tuples).

Using distinct batches requires being able to control to which consumer an output tuple is sent. In our example in Figure 9 each batch corresponding to the first logical consumer A (i.e., the top three batches) is only sent to one of the top three task belonging to the first logical consumer A. Each batch corresponding to the second logical consumer B (the bottom two batches) is sent only to one of the two bottom tasks which belong to the second logical consumer. Hence, an upper batch is never sent to a lower consumer and a lower batch is never sent to an upper consumer. Storm’s default behavior is to
replicate all output tuples to all logical consumers. Using distinct batches requires to disable this default behavior and to send each output tuple (i.e., batch) to a single logical consumer only.

Even if Storm allows to send output tuples to a single logical consumer only, it is more difficult to realize (i.e., more difficult to implement) than using the default replication behavior. Furthermore, there are streaming systems that do not allow controlling to which logical consumer an output tuple/batch is sent. If we would use distinct batches in such a system, each output batch would be sent to two tasks (one of the top three, and one on the bottom two). Therefore, we would get (1) tuple duplicates (one replica for each logical consumer) and (2) incorrect routing decision (similar to the case illustrated in Figure 8). Because we want to develop a technique that can be used in other system as well (and to avoid the more complex implementation of distinct batches), we developed *shared batches* as described in the next section.

Figure 10: Example for shared batches for multiple consumers.

**Shared Batches for Different Consumers**

In order to be able to support batching for multiple logical consumers in the case in which we cannot control to which consumer an output tuple is sent, we need to group tuples into batches in a way such that each batch is routed correctly to all consumers. We call this technique “*shared-batches*” because we use each batch for all consumers. Shared-batches requires more output buffers than distinct-batches (number of tasks of the first consumer times number of tasks of the second consumer). Using shared-batches, two tuples are put into the same output buffer if and only if the hash values of the first key-
attributes (i.e., key-attributes of the first consumer) are the same and the hash values of the second key-attributes (i.e., key-attributes of the second consumer) are the same. Let \( H_1 \) and \( H_2 \) be the two hash functions for the first and second consumer respectively. Two tuples \( t_1 \) and \( t_2 \) are put into the same batch if \( H_1(t_2) = H_1(t_1) \) \( and \) \( H_2(t_1) = H_2(t_2) \). This scheme results in a quadratic number of possible hash combinations (quadratic in the number of tasks for both consumers). Figure 10 illustrates the shared-batches scheme. Having three and two tasks for the consumers A and B respectively, means that six [three times two] buffers are needed. The tasks of consumer A process the tuples (shown as small colored arrows) partitioned by color (blue+red and green and violet+orange) while the tasks of consumer B need the tuples grouped by shape (square+circle and plain). The fat solid arrows indicate which batches are sent to which consumer task of the first logical consumer. The two buffers containing red and blue arrows both go to the same consumer task. However, we need to keep both buffers separated, because we need to separate shaped arrows from plain arrows. Note that the fat dotted line of both buffers goes to different tasks for the second consumer. Hence, both batches go to different tasks for the second consumer.

Recall that the routing (fat arrows in Figure 10) is done by the underlying streaming system and we assume that we cannot influence it. Hence, we prepare the batches in a way (as described above), such that the system will employ the described pattern (i.e., sending each batch to two tasks, each belonging to one logical consumer) automatically. In Figure 9 (showing distinct batches) in contrast, we assume that we are able to control to which (logical) consumer a batch is sent and the system only picks a consumer task (distinct batches also works if we can pick the consumer task directly, but we do not require this). Therefore, the batches for the first consumer are only sent to the first consumer and the batches for the second consumer are only sent to the second one.
Figure 11 shows a flow chart which applies to all batching schemes. The blue part is the unchanged flow while the green part is inserted for batching.

Figure 11: Key-based batching on top of intra-node parallel distributed streaming system.
4. An Implementation Perspective on Storm

In the previous section we gave a conceptual overview about the design of our batching layer. This section will give more implementation details that are Storm specific. In order to understand the details about the architecture of the batching layer, recall the UserSpout and UserBolt code snippets from Section 2. We want to point out that the output tuples for each node in a topology are handed over to Storm by calling the emit() method of the OutputCollector object. Hence, the call hierarchy is as follows (top right part in Figure 12): (1) For Spouts the nextTuple() method is called by Storm in an infinite loop to request new tuples for processing. (2) Each time a new tuple is available the user code makes a back-call to Storm using the OutputCollector.emit() method that puts the tuple into an output queue. An asynchronous thread takes the tuples from this output queue and transfers them to the corresponding consumer task. In order to employ batching, we provide a wrapper for the original OutputCollector and capture the calls to the emit() method (bottom part in Figure 12). Our wrapper buffers multiple tuples until a batch is completed and then performs the back-call to the original OutputCollector emit() method in order to hand the whole batch to Storm. From an implementation point of view, we need a second wrapper class for the producer node itself, too. This wrapper for the producer node is necessary to perform the wrapping of the OutputCollector (the open() method is the only point at which we get access to the OutputCollector). Figure 12 (bottom part) shows the code layers. In the case of batching Storm calls open() not on the original Spout object but on the corresponding batching wrapper (shown in green). Within open() the original OutputCollector is
wrapped and the wrapper is given to the user provided Spout object (1). The wrapper of the Spout forwards the calls, made by Storm to `nextTuple()` by calling `nextTuple()` of the wrapped Spout (2). Each time the user calls `emit()`, it actually calls the wrapper of the OutputCollector (3), that eventually calls the original OutputCollector to emit a batch tuple (4). At the bottom of Figure 12 it is illustrated that the code parts of Storm (in blue) are completely isolated from the user code (red) by two wrapper classes (green). Therefore, it is possible to capture/redirect any⁴ calls made by Storm to the user code and vice versa.

For Bolts the pattern is exactly the same. However, instead of `open()` and `nextTuple()` the corresponding methods `prepare()` and `execute()` are used. Find below code snippets for Spout and Bolt wrappers that perform the wrapping of the OutputCollector within `open()` and `prepare()` respectively. Calls to `nextTuple()` and `execute()` are simply forwarded to the original user code.

```java
public class SpoutOutputBatcher implements IRichSpout {
    private final IRichSpout wrappedSpout;
    private final int batchSize;

    public SpoutOutputBatcher(IRichSpout spout, int batchSize) {
        this.wrappedSpout = spout;
        this.batchSize = batchSize;
    }

    @Override
    public void open(Map conf, TopologyContext context,
                     SpoutOutputCollector collector) {
        // plug-in the SpoutBatchCollector
        // (transparently to the Spout to be wrapped)
        this.wrappedSpout.open(conf, context,
                               new SpoutBatchCollector(context, collector, this.batchSize));
    }

    @Override
    public void nextTuple() {
        this.wrappedSpout.nextTuple();
    }
}

public class BoltOutputBatcher implements IRichBolt {
    private final IRichBolt wrappedBolt;
    private final int batchSize;

    public BoltOutputBatcher(IRichBolt bolt, int batchSize) {
        this.wrappedBolt = bolt;
        this.batchSize = batchSize;
    }

    @Override
    public void prepare(Map stormConf, TopologyContext context,
                        BoltDelegator delegate) {
```
As mentioned above, the wrappers of the OutputCollector capture `emit()` calls, build a batch tuple, and emit it to Storm when it is completed. Those wrappers are using a `BatchCollector` class that uses the `Batch` class as described in Section 2 to buffer the tuples. Following the different batching schemes multiple instances of `Batch` are used—one for each needed output buffer. `BatchCollector` implements all the details described in Section 2 to build the output batches correctly. Details are omitted in the code snippets below:

```java
public class SpoutBatchCollector extends SpoutOutputCollector {
    private final SpoutOutputCollector delegate;
    final BatchCollector batcher;

    public SpoutBatchCollector(TopologyContext context,
    SpoutOutputCollector delegate, int batchSize) {
        super(delegate);
        this.delegate = delegate;
        this.batcher = new BatchCollector(context, batchSize);
    }

    @Override
    public List<Integer> emit(List<Object> tuple) {
        this.batcher.emit(tuple);
    }
}

public class BoltBatchCollector extends OutputCollector {
    private final OutputCollector delegate;
    final BatchCollector batcher;

    public BoltBatchCollector(TopologyContext context,
    OutputCollector delegate, int batchSize) {
        super(delegate);
        this.delegate = delegate;
        this.batcher = new BatchCollector(context, batchSize);
    }

    @Override
    public List<Integer> emit(List<Object> tuple) {
        return this.batcher.emit(tuple);
    }
}
```
The nice property of our wrapping architecture is that neither Storm nor the user’s topology code needs to be adapted. The user only needs to use the wrappers for Spout/Bolt when building the topology. The constructors of these wrappers have an additional parameter that specifies the size of the batch (compare to the code snippets for SpoutOutputBatcher and BoltOutputBatcher above). Two code examples (w/o and w/ batching) are shown below:

```java
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(new UserSpout());

TopologyBuilder builder = new TopologyBuilder();
int batchSize = 5;
builder.setSpout(new SpoutOutputBatcher(new UserSpout(), batchSize));
```

The ability to insert the provided wrappers without changing the user code allows Aeolus\(^5\) to insert the wrapper automatically. Aeolus’ optimizer extends Storm’s TopologyBuilder class. Therefore, the user only needs to modify a single line of code while building a topology and does not even need to insert the wrapper by his/her own. Instead of using Storm’s TopologyBuilder, the user creates a TopologyOptimizer object provided by Aeolus. The remainder of creating a topology can be used unmodified. The example below shows the code for building a topology as in the above example. The user does not insert the Spout wrapper. Aeolus decides if the Spout output should be batched or not. If batching is beneficial, Aeolus also decides on the batch size and inserts SpoutOutputBatcher automatically:

```java
TopologyBuilder builder = new TopologyOptimizer();
builder.setSpout(new UserSpout());
```

**Special Spout Behavior**

Since Storm 0.8.2, Spouts do have a special feature to avoid *busy waiting*. As mentioned earlier, each call to `nextTuple()` or `execute(...)` can result in any number of output tuples (including zero). Emitting no tuples is a common behavior for Bolts, e.g., a Bolt might implement a filter. Even if a Bolt does not output a tuple, it performed useful work because an input tuple was processed. In case of Spouts the situation is a little bit different. Even if emitting no tuples is also common for Spouts (for example, a Spout might fetch data externally, but no new data is available), a Spout did not perform useful work (in contrast to a Bolt). Imagine the case in which a Spout fetches external data from a web source. If the web source is not reachable for whatever reason, the Spout would not emit tuples for some time. In older versions of Storm, this resulted in a *busy wait* situation. Storm calls `nextTuple()` again and again (with a high frequency) but no useful work is performed because no tuples are emitted by the Spout. In order to avoid this busy waiting situation, Storm (since version 0.8.2) applies a waiting penalty before calling `nextTuple()` again if no output tuple was generated by a Spout. This penalty is increased each time emit is not called and set back to zero if emit is called. The pseudo code for the Spout calling strategy is shown below:

---

\(^5\) Aeolus is the name of our optimizer that determines for which nodes in a given topology batching should be used and what the batching size should be. It makes use of our batching layer. Furthermore, Aeolus computes an appropriate dop for each node of a topology. This report does not cover optimization details, i.e., how Aeolus’ optimizer computes the values for dop and batch size [2].
while topology is running
    set oldEmitCount = currentEmitCount
    call nextTuple()
    if currentEmitCount == oldEmitCount
        then
            increase waitingPenalty
            sleep for waitingPenalty
        else
            set waitingPenalty = 0
        endif

This waiting strategy avoids busy waiting successfully and saves unnecessary CPU consumption resulting in better system behavior. However, it does not work well together with our batching layer. If the output of a Spout is subject to be batched, most calls to nextTuple() will not result in an emit call even if an output tuple was produces, because the BatchCollector absorbs most emit calls. Hence, from a Storm perspective no useful work was done and a waiting penalty is applied. Hence, Storm calls emit less frequent as it actually should and the Spout output rate is decreased. We resolve this problem by adapting the call strategy to the original nextTuple() method within our SpoutOutputBatcher. After each call of userSpout.nextTuple() we check if emit was called, i.e., if a tuple was inserted into the output batch. In this case, we call userSpout.nextTuple() again. We break this loop under two conditions: (1) No tuples was inserted into the batch. In this situation, the Spout did not perform useful work and we need to break the loop in order to avoid busy waiting. Storm will apply the waiting penalty in this case what is desired behavior. (2) A batch was completed and emitted. In this second case, we also break the loop and return to Storm. Because a batch was emitted, Storm does not apply the waiting penalty. From a Storm perspective, a single call to nextTuple() resulted in an output tuple. Breaking the loop in this situation is important, because Storm performs some other work after each nextTuple() call. For example, Storm monitors the execution time of each call. If we would not break the loop, Storm would not perform that work and the system behavior would be modified. Furthermore (and from an engineering point of view), our batching layer is actually user code and not system code. Hence, it should not take control but leave control to Storm. Hence, breaking the loop in both cases as described results in desired behavior. The code example below shows the modified code for SpoutOutputBatcher (modified open() and nextTuple()):

```java
public class SpoutOutputBatcher implements IRichSpout {
    private SpoutBatchCollector batchCollector;

    @Override
    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector collector) {
        // plug-in the SpoutBatchCollector
        // (transparently to the Spout to be wrapped)
        this.batchCollector = new SpoutBatchCollector(context, collector,
            this batchSize);
        this.spout.open(conf, context, batchCollector);
    }

    @Override
    public void nextTuple() {
```
while(true) {
    this.batchCollector.tupleEmitted = false;
    this.batchCollector.batchEmitted = false;
    this.spout.nextTuple();
    if (!this.batchCollector.tupleEmitted 
        || this.batchCollector.batchEmitted) {
        break;
    }
} 

We want to point out, that we also need to modify SpoutBatchCollector. We introduce the new variables tupleEmitted and batchEmitted that are set appropriately. Hence, both wrappers (for Spout and OutputCollector) must cooperate to achieve the proper behavior. The modified code for wrapping the OutputCollector is shown below. The variable batchEmitted is set to true by the this. batcher object if a batch is emitted with its emit method (not shown in the code snippet):

```java
public class SpoutBatchCollector extends SpoutOutputCollector {
    boolean tupleEmitted = false;
    boolean batchEmitted = false;
    
    @Override
    public List<Integer> emit(List<Object> tuple) {
        this.tupleEmitted = true;
        // batcher might set this.batchEmitted to true
        return this.batcher.emit(tuple);
    }
}
```

**De-Batching at the Consumer**

We only described how to build (output) batches in the previous sections. However, to be transparent to the user, we also need to perform de-batching at the consumer side (otherwise, the consumer execute-method would be called with batches of tuples, instead of individual tuples). We provide a second wrapper class for Bolts that performs de-batching: InputDebatcher. InputDebatcher is designed s.t. it works independent from BoltOutputBatcher (and vice versa). This enables their usage for all four possible batching/non-batching cases:

1. input not batched, output not batched (none of both wrappers is used)
2. input not batched, output is batched (BoltOutputBatcher is used)
3. input is batches, output is not batcher (InputDebatcher is used)
4. input is batched, output is batched (BoltOutputBatcher and InputDebatcher are used)

InputDebatcher is inserted into a topology similar to Spout/BoltOutputBatcher. Aeolus is able to insert InputDebatcher into a topology automatically in the same way as Spout/BoltOutputBatcher. An example is shown below, including the case that both, input and output, are batched:
TopologyBuilder builder = new TopologyBuilder();
// no output batching for Spout
IRichSpout source = new UserSpout();
builder.setSpout(source);
// output batching for first Bolt
IRichBolt bolt1 = new BoltOutputBatcher(new UserBolt(), 5);
bolt1.shuffleGrouping(source);
// input de-batching AND output batching for second Bolt
IRichBolt bolt2 = new InputDebatcher(new BoltOutputBatcher(new UserBolt(), 5));
bolt2.shuffleGrouping(bolt1);
// input de-batching for third Bolt
IRichBolt bolt3 = new InputDebatcher(new UserBolt());
bolt3.shuffleGrouping(bolt2);

The basic functionality of InputDebatcher is to take an input batch, to extract the original data tuples, and to call execute() for each tuple using the original Bolt object. Because a single Bolt can have multiple predecessors and not all of them might have batched output, it is also necessary to check if the incoming tuple is a regular data tuple or an input batch. Find below the example code:

```java
public class InputDebatcher implements IRichBolt {
    private final IRichBolt wrappedBolt;
    
    public InputDebatcher(IRichBolt bolt) {
        this.wrappedBolt = bolt;
    }

    @Override
    public void execute(Tuple input) {
        if (input instanceof Batch) {
            for (int i = 0; i < input.getBatchSize(); ++i) {
                this.wrappedBolt.execute(input.getNextTuple());
            }
        } else {
            this.wrappedBolt.execute(input);
        }
    }
}
```
5. Batching and Storm’s Fault-Tolerance Mechanism

Storm has a specific fault-tolerance mechanism that guarantees *at-least-once* processing for all tuples emitted by a “reliable” Spout. A Spout is “reliable” if all emitted tuples are stored until they are “completely processed”. If a reliable Spout is used, each tuples get a unique ID and Storm informs the Spout if a tuple was successfully (also called completely) processed or if processing failed. In order to use this mechanism, the user must implement the two Spout methods `ack()` and `fail()`. Storm calls the appropriate method (with a tuple ID as parameter) if processing a tuple was successful or not respectively. In order to keep track of tuple processing, the user-defined code within the Bolts must “link” input and output tuples to each other and “acknowledge” processed tuples. If a tuple is not acknowledged within a timeout, Storm automatically assumes this tuple was not successfully processed (failed). Additionally, the user-code can fail a tuple directly if it cannot be processed for some reason. The tracking of the tuples and linking of input and output tuples uses the concept of a processing tree as described in the next paragraph. In order to keep Storm’s fault-tolerance mechanism working, our batching layer must perform additional work. We explain the needed concepts and their implementation in this section.

Storm’s Processing Tree

Each output tuple that is emitted by a Spout forms a so-called *Processing Tree (PT)*. The PT of a tuple consists of all its child tuples in a recursive manner (i.e., all child tuples of every child and so forth). In order to keep track of the PT, the user needs to link input and output tuples to each other. This mechanism is called anchoring in Storm terminology. Each time the user emits a tuple (in any node) the output tuple can be anchored to all tuples that are necessary to re-compute this output tuple. Anchoring is a very generic process: each output tuple can be anchored to multiple input tuples, e.g., in case of computing an aggregation. At the same time, an input tuple can be used as an anchor for multiple output tuples. For example, a Bolt can implement a sliding window (that is overlapping). If each window slide results in an output tuple, a single input tuple must be anchored to multiple output tuples. In case of a failure, Storm is able to resubmit the necessary input tuples such that the output tuple can be recovered. In order to reprocess tuples, all tuples must be saved reliably until they are not needed anymore. For example, a Spout output tuple can be deleted if the whole PT is completely processed. Because Storm cannot decide automatically if a PT is completed or not, the user need to acknowledge (ack) tuples after processing is finished. If all tuples from a PT are acked, the original input tuples can be deleted safely. It can also happen that a tuple cannot be processed successfully. It this case, the tuple fails. If Storm detects a failed tuples, reprocessing is triggered. Each tuple must either be acked or failed by the user. If the user does not ack a tuple (and does not fail it either), Storm applies a timeout (per default 30 seconds). After this timeout, Storm automatically fails this tuple. We want to point out, that tuple processing includes anchoring, i.e., an input tuple cannot be used as an anchor after it got acked.

Anchoring and Aicking for Output-Batches

The OutputCollector is used by the user to emit, anchor, ack, or fail tuples. Anchoring is done within the `emit()` method. Emit() can have multiple arguments and one can be a list of “parent” (i.e., anchor) tuples for the given output tuple. The emitted output tuple is anchored to all tuples in this anchor list. After an input tuple is completely processed, i.e., it is not needed to produce more output tuples (and it
is not used as an anchor for any other output tuple), the user must ack the tuple. It is important that the call to \texttt{this.collector.ack(inputTuple)} is the last action to perform on a tuple to enable Storm to meet its reliability guarantees, i.e., \texttt{ack(...)} is called \textit{after} \texttt{emit(...)} if the acked tuples is used as an anchor in the emit-call.

If batching is used, the batching layer must take care the instead of regular data tuples batches must be anchored and acknowledge correctly. Because batching in transparent to Storm and to the user code, the user code will anchor tuples to each other and ack tuples that might be unknown to Storm (because Storm is only aware of a batch, but not the tuples contained in the batch). Hence, this user calls must be “redirected” to the corresponding batch tuples.

Let us assume a simple tuple forward example as shown below. Each input tuple is simply forwarded by (deep) copying \texttt{(input.getValues())}. At the same time, each output tuple is anchored to its corresponding input tuple (the first parameter of \texttt{emit} is the anchor tuple). The following code shows the corresponding implementation of a Bolt’s \texttt{execute()} method:

\begin{verbatim}
public void execute(Tuple input) {
    this.out.emit(input, input.getValues());
}
\end{verbatim}

If we apply batching to the output, we need to consider anchoring. Assuming an output batch size of three tuples, we need to anchor the output batch to all three input tuples that are forming this batch. Hence, our batching layer must accumulate all anchor tuples over all \texttt{emit} calls until an output batch is full. Figure 13 illustrates the modified anchoring. Each input tuple \( t_1, t_2, \) and \( t_3 \) results in an output tuple \( t'_1, t'_2, \) and \( t'_3 \). The anchoring is indicated by the red arrows. In case of batching \( t'_1, t'_2, \) and \( t'_3 \) are never ‘seen’ by Storm, but only the batch tuple \( b \). From a Storm point of view \( t'_1, t'_2, \) and \( t'_3 \) do not even exist. At the same time, we need all three input tuples to recreate the batch tuple. Hence, we anchor the batch to all three input tuples.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure13.png}
\caption{Anchoring w/o and w/ batching.}
\end{figure}
Batching and anchoring the tuples as describes is not enough though. We also need to take care of acking. Let’s assume to have a batch size of three tuples. Let the user code emits and acknowledges each input tuple as shown (this is again a simple tuple forward):

```java
public void execute(Tuple input) {
    this.out.emit(input, input.getValues());
    this.out.ack(input);
}
```

Because the OutputCollector is wrapped with our BatchCollector, the output tuples are actually not given to Storm. Hence, the acknowledgment of the tuples breaks Storm’s fault-tolerance guarantees (acking must happen after anchoring, but anchoring is “delayed” within the batching layer). If a Bolt fails after processing the first input tuple, the batch was not emitted. Hence, the output tuple is lost. Because of their acknowledgment, Storm will not replay the corresponding input tuples. In order to resolve this problem, the BatchCollector must also hide the acks from Storm until the batch is full and was emitted, i.e., the BatchCollector must do the acks after emitting the batch (Figure 14, highlighted). We accomplish this behavior by keeping a list of tuples the user acked and ‘cancel’ the actual ack-calls (by not forwarding them to Storm). Each time a batch is emitted, all tuples in the build ack-tuple-list are acked to Storm. Recall the example from above in which two output tuples got lost. In the case of the improved BatchCollector both tuples are not acknowledged. Therefore, Storm replays the corresponding input tuples. (Acking tuples does only happy for Bolts and not for Spouts because Spouts do not have input tuples. Therefore, only BoltBatchCollector has the appropriate code.) Figure 14 shows the call

![Figure 14: Acking of input tuples w/o and w/ batching.](image-url)
trace of emit and ack calls for the regular (non-batching) case on the left (example from Figure 13). The batching case is shown on the right. The first two emit calls are captured and the output tuples are inserted into the batch buffer. The corresponding ack calls are “remembered” but not forwarded to Storm. The first emit call complete the batch, triggers the emit of the batch (including the anchoring to the three input tuples), and the replay of the two stored ack calls. Finally, the first ack call is forwarded to Storm immediately.

Pay attention, that tuple t3 from Figure 13 is not acked right away after emitting the batch tuple. The user’s emit call that completes the output batch happens before the user’s ack call for t3. Hence, the BatchCollector cannot ack t3 (maybe the user does not ack t3 at all...). In our example, t3 will be acked when the user acks t3. BoltCollector recognizes that the corresponding output batch of t3 is already emitted to Storm. Hence, the ack call must not be delayed but can be forwarded to Storm immediately. Example code for batch collector and Bolt wrappers is shown below:

```java
public class BoltBatchCollector extends OutputCollector {
    private final OutputCollector delegate;
    final BatchCollector batcher;

    public BoltBatchCollector(TopologyContext context,
                               OutputCollector delegate, int batchSize) {
        super(delegate);
        this.delegate = delegate;
        this.batcher = new BatchCollector(context, batchSize);
    }

    @Override
    public List<Integer> emit(Collection<Tuple> anchors, List<Object> tuple) {
        // batcher will anchor the output batch to all anchor tuples
        // batcher will replay acks that are buffered within .ack()
        this.batcher.emit(anchors, tuple);
    }

    @Override
    public void ack(Tuple input) {
        // remember that user acked this tuple if output batch is not full
    }
}
```

**Anchoring for Input Batches**

In the previous section we pointed out, that the anchoring behavior must be adapted for output batches. The same is true for input batches. Assume a Bolt that receives input batches while the output is not batches. InputDebatcher extracts data tuples from the input batch by creating new tuple objects. From a Storm point of view, those (internal) tuples do not exist. However, the user might use those tuples as anchors for output tuples. The user is not aware that batching is used and that the given input tuples are no known by the system and would result in ‘dangling’ anchors. Because anchoring describes tuple dependencies, we need to redirect the anchoring of the output tuples to the input batch tuple. If any output tuple fails, we need to reprocess the whole batch to be able to recover the lost output tuple.
Figure 15 show the correct anchoring that is performed by the InputDebatcher. The anchor relationship is shown by the red arrows.

Because anchoring is performed at the OutputCollector, InputDebatcher needs to wrap the OutputCollector in the same way as OutputBatcher did (i.e., we introduce a new wrapper class). Within the prepare(...) method the OutputCollector is wrapped and the wrapper is given to the user code. Both wrapper classes (InputDebatcher and the wrapper of OutputCollector) must cooperate to achieve the desired behavior (in opposite to anchoring for output batches where solely the BoltOutputCollector is responsible for redirecting anchors and acks). We use a map that links all internally created data tuples to their corresponding batch tuple as a data structure (done by InputDebatcher). Each time the user calls emit(), the call is capture and it is checked if the provided anchor tuples are original system tuples known by Storm or internally created tuples. If an anchor tuple is an internal tuple, it is replaced by its corresponding batch tuple (using the tuple-to-batch-map) within the list of anchor tuples (done by OutputCollector wrapper). The check if an anchor tuple is an original tuple or an internal tuple is necessary because a Bolt can have multiple predecessors and not all of them might batch their output. This check is similar to the check described in Section 3 that InputDebatcher does before trying to perform de-batching. Find below example code for InputDebatcher and the new AnchoringWrapper:

```java
public class InputDebatcher implements IRichBolt {
    private final IRichBolt wrappedBolt;
    final Map<Tuple, Tuple> internalTupleToBatch = new HashMap<Tuple, Tuple>();
    
    public InputDebatcher(IRichBolt bolt) {
        this.wrappedBolt = bolt;
    }

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        // plug-in the AnchoringWrapper
        // (transparently to the Bolt to be wrapped)
        this.bolt.prepare(stormConf, context,
    }
```
In order to avoid an infinite grows of the user map that links tuples to their batches, we do remove entries in the map when the user acks or fails an internal tuple. After acking/failing a tuple, the user code in not ‘allowed’ to use this tuple for anchoring anymore to meet Storm’s fault-tolerance guarantees. Hence, we can remove the entry in the map. If the user still does use an internal tuple for anchoring after ack/fail, the anchor will be ‘dangling’ and not be valid. But the anchor would not be valid in the non-batching case, too. Therefore, we do not break Storm’s fault-tolerance model with this strategy.

### Tuple Acknowledgement for Input Batches

In this section, we describe how to handle tuple acknowledgments for input batches. As described above, de-batching creates internal tuples that are not known by Storm, but those tuples are used by the user as if they would be regular tuples. If the user acks or fails an internal tuple, we need to redirect the call to the corresponding input batch. In contrast to anchoring however, we cannot simply ack the batch tuple if an internal tuple is acked. Let the input batch size be three tuples. Assume that the user processed and acked the first tuple from the input batch. If we ack the input batch right away, but the Bolt fails before processing the second and third tuple, those corresponding output tuples are lost.
Because the input batch is already acked to Storm, it will not be replayed. Therefore, we need to wait before all input tuples are processed by the user before we can ack the input batch, i.e., the user needs to ack all internal tuples before we can ack the batch tuple. Similar to the anchoring strategy, InputDebatcher and the wrapper of the OutputCollector must cooperate to achieve the desired behavior. We introduce a second data structure within InputDebatcher that is shared with the output wrapper. This data structure is a map that keeps a list of all internal tuples that are not acked/fail by the user for each input batch. Each time, InputDebatcher receives a new batch tuple it creates an entry in the map and inserts all created internal tuples into the corresponding list for the input batch (performed within `execute(...)`). Each time the user acks in internal tuple, the OutputCollector wrapper removes the tuple from this list. After InputDebatcher looped over the whole input batch and called `userBolt.execute(...)` for each internal tuple, it checks if the user acked all internal tuples (i.e., checks if the list of unacked tuples is empty). If this is the case, the input batch gets acked to Storm and the map entry is removed. Because, the user can ack internal tuples at any point in time it can happen that an input batch is not acked immediately after processing all internal tuples. Therefore, input batches can remain in the map. At the same time, the user can ack `old` internal tuples at any point in time, too. Hence, it can happen that the user acks internal tuples from old input batches. Because old input batches are already completely processed, they can be acked to Storm directly after the last internal tuple got acked. Therefore, we check within each ack call, if this tuple completes the acking of an input batch (i.e., if all tuples from an input batch got acked by the user). If an input batch gets completed, our wrapper acks the input batch to Storm. Hence, each input batch will be acked to Storm eventually as along as the user acks all internal tuples.

If the user fails a tuple, we do not need to delay the fail but we can fail the input batch immediately. Additionally, we break the loop within InputDebatcher that processed the input batch. Because the batch will be replayed by Storm anyway, there is no need to finish processing the batch. This strategy improves performance and avoids producing unnecessary duplicate tuples (if we would keep processing, all remaining tuples in the input batch would be replayed when the complete batch is replayed, possibly resulting in duplicate output tuples). Find below the modified code for InputDebatcher (modified `execute(...)` and AckFailAnchoringWrapper (originally called AnchoringWrapper; modified `ack()` and `fail()`):

```java
public class InputDebatcher implements IRichBolt {
    private final IRichBolt wrappedBolt;
    final Map<Tuple, Tuple> internalTupleToBatch = new HashMap<Tuple, Tuple>();
    final Map<Tuple, List<Tuple>> notAckedTuplesPerBatch = new HashMap<Tuple, List<Tuple>>();

    public InputDebatcher(IRichBolt bolt) {
        this.wrappedBolt = bolt;
    }

    @Override
    public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
        // plug-in the AnchoringWrapper
        // (transparency to the Bolt to be wrapped)
    }
```
this.bolt.prepare(stormConf, context, new AckFailAnchoringWrapper(collector));

@Override
public void execute(Tuple input) {
    if (input instanceof Batch) {
        for (int i = 0; i < input.getBatchSize(); ++i) {
            t = input.getNextTuple();
            // link internal tuple to input batch
            this.internalTupleToBatch.put(t, input);
            // add internal tuple list of unacked tuples
            this.internalTupleToBatch.get(input).add(t);
            this.wrappedBolt.execute(t);
        }
        // check if all internal tuple got acked by user
        if (this.notAckedTuplesPerBatch.get(input.size()) == 0) {
            // all internal tuple are acked
            // -> ack input batch, remove batch entry
            this.collector.ack(input);
            this.notAckedTuplesPerBatch.remove(input);
        } else {
            this.wrappedBolt.execute(input);
        }
    }
}

public class AckFailAnchoringWrapper extends OutputCollector {
    private final OutputCollector delegate;

    public AnchoringWrapper(OutputCollector delegate) {
        super(delegate);
        this.delegate = delegate;
    }

    @Override
    public void ack(Tuple tuple) {
        if (input instanceof Batch) {
            // tuple is acked, so we can remove batch---anchoring is done
            this.debatcher.internalTupleToBatch.remove(tuple);
            // remove tuple from list of unacked tuples
            this.debatcher.notAckedTuplesPerBatch.get(batch).remove(tuple);
            // check if old input batch is completed
            // i
        } else {
            // tuple is regular tuple -> forward ack to Storm
            this.delegate.ack(tuple);
        }
    }

    @Override
    public List<Integer> emit(Collection<Tuple> anchors, List<Object> tuple) {
        // checkAnchors(...) checks for each anchor in the list,
        // if it is an original or internal tuple
        // internal tuple are replace by batchTuples
Wrapping a Bolt with an InputDebatcher (and wrapping the corresponding OutputCollector) results in a call hierarchy similar to the case for batching output at a Spout or Bolt. Figure 16 illustrates the call hierarchy for the input batches. Again, the user code (red) is completely encapsulates in out batching layer (green). Storm (blue) cannot access the user Bolt directly and vice versa (i.e., no direct back-calls to Storm).

**Figure 16: Call hierarchy of Storm, Bolts, and OutputCollector w/ de-batching wrappers.**

Processing Input and Output Batches at once

In the previous paragraphs we described how to batch output tuples and how to deal with incoming batch tuples independently. In this section, we describe the interaction of both approaches together. Because we build the wrappers for both cases transparently, we can use both at once. Code examples from above showed already cases in which a Bolt is wrapped twice. In this example, the user Bolt is first wrapped with the OutputBatcher and then further wrapper with an InputDebatcher. This wrapping order is critical in the current implementation. A reverse wrapper order is not supported.
6. Conclusion
In this report, we gave an overview about Storm’s system architecture, programming, and execution model. We explained the general idea and basic design of batching in stream processing systems. Furthermore, we suggested different batching schemes, namely “distinct batches” and “shared batches”, which can be used within intra-node-parallel stream processing system to implement batching correctly. We gave a details description of the architecture of our transparent batching layer – including code snippets – and explained how we preserve Storm’s fault-tolerance guarantees.

In the appendix, we give some examples that show how Storm and our batching layer are interacting with each other. We did not include an evaluation of performance improvements that we achieve with batching and refer the interested reader to our papers [2,3] that covers Aeolus’ optimization approach including performance results.

References
[1] Storm web site: http://storm-project.net/


Appendix

Storm-Aeolus-Interaction

In the following, we show the calls traces from Storm to a topology and from the topology back to Storm using different input and output batch sizes. The used topology consists of a single Spout and Bolt. The Spout emits tuples with increasing ID (show in brackets) at each call on `nextTuple()` while the Bolt performs a simple tuple forwarding as shown previously. Pay attention, that the shown calls interleave: for example if an call of `execute(...)` is followed by `emit(...)` it does not imply that `execute(...)` finished already. In contrast, all calls that follow `execute(...)` are triggered by it (up to the next `execute-call`). To keep the examples simple, we do not show the calls made by the batching layer internally.

No Batching

If no batching is used, each call of `nextTuple()` results in a back-call of `emit(...)` for the Spout as shown below:

```
Spout.nextTuple()
SpoutOutputCollector.emit(): [1]
Spout.nextTuple()
SpoutOutputCollector.emit(): [2]
Spout.nextTuple()
SpoutOutputCollector.emit(): [3]
Spout.nextTuple()
SpoutOutputCollector.emit(): [4]
Spout.nextTuple()
SpoutOutputCollector.emit(): [5]
```

The `execute(...)` method at the Bolt is called for each incoming tuple, resulting an one `emit(...)` and one `ack(...)` back-call to Storm:

```
Bolt.execute(): [1]
OutputCollector.emit(): [1]
OutputCollector.ack(): [1]
Bolt.execute(): [2]
OutputCollector.emit(): [2]
OutputCollector.ack(): [2]
Bolt.execute(): [3]
OutputCollector.emit(): [3]
OutputCollector.ack(): [3]
Bolt.execute(): [4]
OutputCollector.emit(): [4]
OutputCollector.ack(): [4]
Bolt.execute(): [5]
OutputCollector.emit(): [5]
OutputCollector.ack(): [5]
```

Spout Output Batching / No Bolt Output Batching

If the output of the Spout is batched with a batch size of 5 tuples, each call of `nextTuple()` results in a single back-call as well. However, the emitted tuple is a whole batch of 5 tuples instead of a single
tuple. Internally, our batching layer calls the original `userSpout.nextTuple()` method five times (not shown):

```java
Spout.nextTuple()
SpoutOutputCollector.emit(): [[1, 2, 3, 4, 5]]
Spout.nextTuple()
SpoutOutputCollector.emit(): [[6, 7, 8, 9, 10]]
```

At the consumer, `execute(...)` is called with a batch tuple, resulting in multiple `emit(...)` calls (in our example five because the input batch size is five) and a final `ack(...)` call that acknowledges the batch tuple. Internally, our batching layer extracts the five tuples from the input batch and calls `userBolt.execute(...)` for each of them. The five `ack(...)` calls from the user Bolt are captured resulting in the final acknowledgment of the input batch:

```java
Bolt.execute(): [[1, 2, 3, 4, 5]]
OutputCollector.emit(): [1]
OutputCollector.emit(): [2]
OutputCollector.emit(): [3]
OutputCollector.emit(): [4]
OutputCollector.emit(): [5]
OutputCollector.ack(): [[1, 2, 3, 4, 5]]
Bolt.execute(): [[6, 7, 8, 9, 10]]
OutputCollector.emit(): [6]
OutputCollector.emit(): [7]
OutputCollector.emit(): [8]
OutputCollector.emit(): [9]
OutputCollector.emit(): [10]
OutputCollector.ack(): [[6, 7, 8, 9, 10]]
```

### Only Bolt Output Batching

In this example, the Spout behavior is similar to the case in Section 4.1:

```java
Spout.nextTuple()
SpoutOutputCollector.emit(): [1]
Spout.nextTuple()
SpoutOutputCollector.emit(): [2]
Spout.nextTuple()
SpoutOutputCollector.emit(): [3]
Spout.nextTuple()
SpoutOutputCollector.emit(): [4]
Spout.nextTuple()
SpoutOutputCollector.emit(): [5]
Spout.nextTuple()
SpoutOutputCollector.emit(): [6]
Spout.nextTuple()
SpoutOutputCollector.emit(): [7]
Spout.nextTuple()
SpoutOutputCollector.emit(): [8]
Spout.nextTuple()
SpoutOutputCollector.emit(): [9]
Spout.nextTuple()
```
SpoutOutputCollector.emit(): [10]
Spout.nextTuple()
Spout.nextTuple()
SpoutOutputCollector.emit(): [12]

However, the Bolt uses output batches of size five. Therefore, the first emit(...) call happens after five execute(...) calls because the Bolt emits batches of five tuples. After the batch is emitted, the input tuples are acknowledged. We want to point out, that the ack of the first four tuples is triggered by the fifth user Bolt emit call that results in emitting the batch tuple. The fifth ack it triggered by the user ack call of input tuple five. Because the output batch tuple is already emitted, this fifth ack call is not buffered but forwarded to Storm immediately.

Bolt.execute(): [1]
Bolt.execute(): [2]
Bolt.execute(): [3]
Bolt.execute(): [4]
Bolt.execute(): [5]
OutputCollector.emit(): [[1, 2, 3, 4, 5]]
OutputCollector.ack(): [1]
OutputCollector.ack(): [2]
OutputCollector.ack(): [3]
OutputCollector.ack(): [4]
OutputCollector.ack(): [5]
Bolt.execute(): [6]
Bolt.execute(): [7]
Bolt.execute(): [8]
Bolt.execute(): [9]
Bolt.execute(): [10]
OutputCollector.emit(): [[6, 7, 8, 9, 10]]
OutputCollector.ack(): [6]
OutputCollector.ack(): [7]
OutputCollector.ack(): [8]
OutputCollector.ack(): [9]
OutputCollector.ack(): [10]

**Spout Output Batching and Bolt Output Batching**

In the last case, both Spout and Bolt batch their output. Hence, each call of next tuple result in an emit of a whole batch tuple and the Bolt also emits batches. Depending on the size of the Spout and Bolt batches, the time at which an ack for an input batch happens varies. We distinguish two cases:

1. Spout batch size < Bolt batch size (i.e., Bolt input batch size < Bolt output batch size)
2. Spout batch size >= Bolt batch size (i.e., Bolt input batch size >= Bolt output batch size)

As an example for the first case, consider a Spout batch size of five and a Bolt batch size of seven. After processing the first input batch of five tuples, the output batch is not full and not emit-call is performed. Additionally, the input batch cannot be acked, because the output batch is not emitted yet. While processing the second input batch, the output batch gets completed and emitted, resulting in the ack of the first input batch that is already completely processed. The third input batch completes the second output batch and triggers the ack of the second input batch. In order to complete the third output batch
(tuples 15 to 21) we need to get two more input batches (16 to 20 and 21 to 25). Because this output batch consists of tuples from three input batches, two input batches get acknowledged after the emit (ack of 11 to 15 and 16 to 20). After processing two more input batches the pattern starts to repeat because seven input batches with five input tuples each, resulted in five output batches with seven tuples each (i.e., after the seventh input batch is processed, the fifth output batch is emitted, all input batches are acked, and no “dangling” output batch is present).

Bolt.execute(): [[1, 2, 3, 4, 5]]
Bolt.execute(): [[6, 7, 8, 9, 10]]
OutputCollector.emit(): [[1, 2, 3, 4, 5, 6, 7]]
OutputCollector.ack(): [[1, 2, 3, 4, 5]]
Bolt.execute(): [[11, 12, 13, 14, 15]]
OutputCollector.emit(): [[8, 9, 10, 11, 12, 13, 14]]
OutputCollector.ack(): [[6, 7, 8, 9, 10]]
Bolt.execute(): [[16, 17, 18, 19, 20]]
Bolt.execute(): [[21, 22, 23, 24, 25]]
OutputCollector.emit(): [[15, 16, 17, 18, 19, 20, 21]]
OutputCollector.ack(): [[11, 12, 13, 14, 15]]
OutputCollector.ack(): [[16, 17, 18, 19, 20]]
Bolt.execute(): [[26, 27, 28, 29, 30]]
OutputCollector.emit(): [[22, 23, 24, 25, 26, 27, 28]]
OutputCollector.ack(): [[21, 22, 23, 24, 25]]
Bolt.execute(): [[31, 32, 33, 34, 35]]
OutputCollector.emit(): [[29, 30, 31, 32, 33, 34, 35]]
OutputCollector.ack(): [[26, 27, 28, 29, 30]]
OutputCollector.ack(): {}, [[31, 32, 33, 34, 35]]

For the second case, consider an input batch size of seven and an output batch size of five. The first input batch immediately triggers the emit of the first output batch. Because the input batch is only partly processed at this point in time, it is not acknowledged. After completing the first input batch, there is an uncompleted output batch (containing tuples six and seven). Therefore, the first input batch still cannot be acked. The second input batch (tuples 8 to 14) complete the second output batch and result in the ack of the first input batch (after emitting the second output batch, it is valid to ack the first input batch). The complete call trace is shown below. As in the example above, the pattern repeats after processing 35 tuples (five input batches each with seven input tuples, resulted in seven output batches each five tuples big).

Bolt.execute(): [1, 2, 3, 4, 5, 6, 7]
OutputCollector.emit(): [[1, 2, 3, 4, 5]]
Bolt.execute(): [[8, 9, 10, 11, 12, 13, 14]]
OutputCollector.emit(): [[6, 7, 8, 9, 10]]
OutputCollector.ack(): [[1, 2, 3, 4, 5, 6, 7]]
Bolt.execute(): [[15, 16, 17, 18, 19, 20, 21]]
OutputCollector.emit(): [[11, 12, 13, 14, 15]]
OutputCollector.ack(): [[8, 9, 10, 11, 12, 13, 14]]
OutputCollector.emit(): [[16, 17, 18, 19, 20]]
Bolt.execute(): [[22, 23, 24, 25, 26, 27, 28]]
OutputCollector.emit(): [[21, 22, 23, 24, 25]]
OutputCollector.emit(): [[15, 16, 17, 18, 19, 20, 21]]
Bolt.execute(): [[29, 30, 31, 32, 33, 34, 35]]
Input and Output Batching with Aggregation

In the previous example, we used a Bolt that forwards each incoming tuple. This example basically describes a tuple-by-tuple processing (similar to a filter or a map). In contrast to the above example, we are now using an aggregation Bolt. To keep the example simple, we consider non-overlapping consecutive groups of five input tuples that are aggregated. The result of the aggregate function is the tuple with the highest ID. The call trace for the non-batching case is shown below. The first output tuple is produced after five execute calls. Because of the aggregate, the first four execute calls cannot result in an output tuple, because the aggregation group of five tuples is not completed. At the same time, none of the input tuples is acked before the emit, because the user cannot ack the tuples before the aggregate was computed. In fact, all five input tuples are used as anchors for the output tuple. Hence, the user-ack for each input tuple happens after the emit.

Spout Output Batching / No Bolt Output Batching

If we use batching for the Spout output with a batch size of seven, the following call trace is the result. First, the Bolt receives an input batch with tuples five to seven. While processing the batch, the first aggregation group is completed, resulting in an emit of tuple five similar to the non-batching case. The user also acks the first five internal tuples. However, those acks are captured by the batching layer because the input batch is not completely processed so far. After the first input batch is completed, the second aggregation group is not completed. Thus, the next step is a second execute call for the Bolt, receiving input batch number two. After tuples eight to ten are processed, the second emit happens.
This emit also triggers the ack of the first input batch. The first input batch is used as an anchor for the first and second Bolt output tuple. Hence, input batch one could not be acked before the second aggregation group was finished resulting in the emit of the second output tuple. The pattern of this call trace is similar to the pattern of the “forward”-example using input batch size seven and output batch size five. However, the output in the forward case are batch tuples, while in the agg case the output are single tuples. Furthermore, the delay of the acks occurs in the user code in the agg example and not in the batching layer. Because the agg-group size is five tuples (equal to the output batch size of the “forward”-example) the call trace pattern starts over after 35 tuples are processed. The processing of input batch $[[29, 30, 31, 32, 33, 34, 35]]$ triggers two emits and two acks (including the ack of the current input batch). Hence, after this step, all input batches are completely processed and acked and no pending aggregation groups are buffered in the user Bolt.

Bolt.execute(): $[[1, 2, 3, 4, 5, 6, 7]]$
OutputCollector.emit(): [5]
Bolt.execute(): $[[8, 9, 10, 11, 12, 13, 14]]$
OutputCollector.emit(): [10]
OutputCollector.ack(): $[[1, 2, 3, 4, 5, 6, 7]]$
Bolt.execute(): $[[15, 16, 17, 18, 19, 20, 21]]$
OutputCollector.emit(): [15]
OutputCollector.ack(): $[[8, 9, 10, 11, 12, 13, 14]]$
OutputCollector.emit(): [20]
Bolt.execute(): $[[22, 23, 24, 25, 26, 27, 28]]$
OutputCollector.emit(): [25]
OutputCollector.ack(): $[[15, 16, 17, 18, 19, 20, 21]]$
Bolt.execute(): $[[29, 30, 31, 32, 33, 34, 35]]$
OutputCollector.emit(): [30]
OutputCollector.ack(): $[[22, 23, 24, 25, 26, 27, 28]]$
OutputCollector.emit(): [35]
OutputCollector.ack(): $[[29, 30, 31, 32, 33, 34, 35]]$
Bolt.execute(): $[[36, 37, 38, 39, 40, 41, 42]]$
OutputCollector.emit(): [40]
Bolt.execute(): $[[43, 44, 45, 46, 47, 48, 49]]$
OutputCollector.emit(): [45]
OutputCollector.ack(): $[[36, 37, 38, 39, 40, 41, 42]]$
Bolt.execute(): $[[50]]$
OutputCollector.emit(): [50]
OutputCollector.ack(): $[[43, 44, 45, 46, 47, 48, 49]]$
OutputCollector.ack(): $[[50]]$

**Only Bolt Output Batching**

In this example, we do not use batching for the Spout. The Bolt batch size is seven. Therefore, the call trace starts with 35 calls of the Bolt execute method. The reason is, that the batching layer captures the emits and the ack of the input tuples until the first output batch is full. Because of the aggregation in the Bolt, multiple input tuples result in a single output tuple. Therefore, the first output batch is completed after seven input groups (i.e., 35 input tuples because of a group size of five tuples) got processed. After the output batch is emitted, the batching layer can replay all the acks of the input tuples that got captures so far. This pattern repeats over and over again. The call trace show 35 more execute calls,
resulting in the emit of a second output batch and another 35 acks that are replayed by the batching layer.

Bolt.execute(): [1]
Bolt.execute(): [2]
Bolt.execute(): [3]
Bolt.execute(): [4]
...
...
...
Bolt.execute(): [32]
Bolt.execute(): [33]
Bolt.execute(): [34]
Bolt.execute(): [35]
OutputCollector.emit(): [[5, 10, 15, 20, 25, 30, 35]]
OutputCollector.ack(): [1]
OutputCollector.ack(): [2]
OutputCollector.ack(): [3]
OutputCollector.ack(): [4]
...
...
...
OutputCollector.ack(): [31]
OutputCollector.ack(): [32]
OutputCollector.ack(): [33]
OutputCollector.ack(): [34]
OutputCollector.ack(): [35]
Bolt.execute(): [36]
Bolt.execute(): [37]
Bolt.execute(): [38]
Bolt.execute(): [39]
...
...
...
Bolt.execute(): [67]
Bolt.execute(): [68]
Bolt.execute(): [69]
Bolt.execute(): [70]
OutputCollector.emit(): [[40, 45, 50, 55, 60, 65, 70]]
OutputCollector.ack(): [36]
OutputCollector.ack(): [37]
OutputCollector.ack(): [38]
OutputCollector.ack(): [39]
OutputCollector.ack(): [40]
...
...
...
OutputCollector.ack(): [67]
OutputCollector.ack(): [68]
OutputCollector.ack(): [69]
OutputCollector.ack(): [70]
Spout Output Batching and Bolt Output Batching

Finally, we are showing an example using a Spout batch size of seven and a Bolt batch size of five tuples. The aggregation is still using a group size of five tuples. Therefore, the call trace starts with four calls of `execute` (each having an input batch of seven tuples). While the first three input batches are processed, four output tuples (5, 10, 15, and 20) are generated and captured by the batching layer. All the internal acks, are also captured such that the first three batch tuples are ready to be acked because all their internal tuples got acked. However, the acks are not forwarded to Storm because the batch tuple will be used as anchors for the first output tuple (all of them contributed to the output). While the fourth input batch is processed, the first output batch is completed. Processing input tuple 25 completes the output batch, resulting in the emit of the first output batch tuple `[[5, 10, 15, 20, 25]]` and the ack of first three input batches. The fourth input batch cannot be acked, because it is not completely processed yet. After the fourth input batch is processed, four more execute calls are happening. During the earlier calls, the next output batch is built up and all acks are captured by the batching layer again. Processing input batch `[[50, 51, 52, 53, 54, 55, 56]]`, i.e., input tuple 50, completes the tenth aggregation group and therefore the second output batch `[[30, 35, 40, 45, 50]]`. The batch tuple gets emitted and the available acks of input batch tuples are replayed by the batching layer.

```plaintext
Bolt.execute(): [[1, 2, 3, 4, 5, 6, 7]]
Bolt.execute(): [[8, 9, 10, 11, 12, 13, 14]]
Bolt.execute(): [[15, 16, 17, 18, 19, 20, 21]]
Bolt.execute(): [[22, 23, 24, 25, 26, 27, 28]]
OutputCollector.emit(): [[5, 10, 15, 20, 25]]
OutputCollector.ack(): [[1, 2, 3, 4, 5, 6, 7]]
OutputCollector.ack(): [[8, 9, 10, 11, 12, 13, 14]]
OutputCollector.ack(): [[15, 16, 17, 18, 19, 20, 21]]
Bolt.execute(): [[29, 30, 31, 32, 33, 34, 35]]
Bolt.execute(): [[36, 37, 38, 39, 40, 41, 42]]
Bolt.execute(): [[43, 44, 45, 46, 47, 48, 49]]
Bolt.execute(): [[50, 51, 52, 53, 54, 55, 56]]
OutputCollector.ack(): [[22, 23, 24, 25, 26, 27, 28]]
OutputCollector.ack(): [[29, 30, 31, 32, 33, 34, 35]]
OutputCollector.ack(): [[36, 37, 38, 39, 40, 41, 42]]
OutputCollector.ack(): [[43, 44, 45, 46, 47, 48, 49]]
Bolt.execute(): [[57, 58, 59, 60, 61, 62, 63]]
Bolt.execute(): [[64, 65, 66, 67, 68, 69, 70]]
Bolt.execute(): [[71, 72, 73, 74, 75, 76, 77]]
OutputCollector.emit(): [[55, 60, 65, 70, 75]]
OutputCollector.ack(): [[50, 51, 52, 53, 54, 55, 56]]
OutputCollector.ack(): [[57, 58, 59, 60, 61, 62, 63]]
OutputCollector.ack(): [[64, 65, 66, 67, 68, 69, 70]]
Bolt.execute(): [[78, 79, 80, 81, 82, 83, 84]]
Bolt.execute(): [[85, 86, 87, 88, 89, 90, 91]]
Bolt.execute(): [[92, 93, 94, 95, 96, 97, 98]]
OutputCollector.ack(): [[80, 85, 90, 95]]
OutputCollector.ack(): [[71, 72, 73, 74, 75, 76, 77]]
OutputCollector.ack(): [[78, 79, 80, 81, 82, 83, 84]]
OutputCollector.ack(): [[85, 86, 87, 88, 89, 90, 91]]
```