SEGMENT BACKUP: DATACENTER-DISASTER TOLERANCE FOR STREAM PROCESSING SYSTEMS

BY

JUANLI SHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Marianne Winslett

ABSTRACT

We design a datacenter-disaster tolerance solution called *segment backup* for stream processing systems. During regular running, segment backup inserts barriers into the normal tuple stream to indicate the backup versions and ensure node-level synchronization of multiple input streams. Additionally, segment backup materializes the partial results at some intermediate nodes to reduce reprocessing work after failures. The simple restarting logic can construct global consistency by connecting the segment consistency with the help of the materialized partial results. This method gives each segment flexibility to roll back to its latest version. We implement a prototype stream processing system with segment backup. Our experimental results show that segment backup can shorten the recovery delay with acceptable overhead during normal execution.

To my parents, for their love and support.

ACKNOWLEDGMENTS

In addition to Professor Marianne Winslett, this project is also supported by Zhenjie Zhang and Tom Fu. They have given me significant help by inspiration, advice, resource and solutions. I appreciate their help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

FAULT TOLERANCE OF STREAM PROCESSING SYSTEMS

1.1 The Basics of Stream Processing Systems

Stream processing systems are computing systems that are usually long-running and real-time. They are different from batch processing systems that usually handle particular complex jobs when required and concern more on throughput. Because of the unique features of stream processing systems, they are used in a lot of monitoring applications, such as network monitoring, stock market monitoring, and social network topic trend monitoring. There are many different stream processing systems nowadays to use. For example, Apache Storm[1], Apache Flink[2], Storm Streaming[3], Samza[4] and etc. Using the terms of Storm, we describe the basics of a stream processing system below.

The story begins with the spouts, who continuously generate the data for later computation. Each unit of the generated data is called a tuple, which is just a list of key/value pairs. The tuples then flow to the next node(s) to be filtered, transferred or joined. Those nodes which operate some specific functions are called bolts. The tuples end at sink nodes, where they may be stored or displayed. The nodes and the tuple stream together form a directed acyclic graph (DAG), which is expressly called a topology.

1.2 Fault Tolerance Requirement

Fault tolerance is a forever pursuit of a system. Since distributed systems are usually set up by commodity servers, failures happen commonly. For stream processing systems, it is even more important, because they often burden real-time monitoring duty. Imagine how much a financial company would lose if its stock tracking system is down for a single minute.

Most of the previous related work focus on the single node failures. When a single node fails, the recovered node or a new node will carry on, hopefully quickly, so that the system can continue working. If the computation is stateful, e.g., accumulating a number, the nodes should back up the corresponding state during normal running and load the state when a new instance is started after a failure.

But what if the whole application fails, or even the entire datacenter in which the application is being run is under disaster? It is required that the whole application can be recovered (restarted), sometimes even in a different datacenter. If we aim to make a restart across datacenters correct and fast, the straightforward idea is to back up whatever can be backed up or even running an active standby, which is super conservative. But the time and space cost would also be extremely high. For long-running systems, cost here means a significant amount of monetary value. We thus try to find a fault tolerance solution that can minimize the overhead cost of a stream processing system while ensuring a decent speed of restarting it across data-centers when application-wide failure happens.

CHAPTER 2

RELATED WORK

2.1 Global Snapshot

When the level of fault tolerance is raised from single-node scenario to whole-application scenario, an essential complexity added on is the consistency. Combining per-node fault tolerance together is easy to say, but maintaining consistency of the whole application and guarantee exactly-once semantics among failures and recovery is difficult to achieve. That is mainly a result of the lack of any accurate global clock and the existence of in-flight messages in the channels, by the nature of the distributed systems. This scenario is where global snapshots should come on the scene to play a role.

Global snapshot is an algorithm that can take a *consistent* snapshot of the global state of a distributed system. The global state should include the states of both the node and the channels. If we can take a global snapshot of our running stream processing application, we can at least restart it with a consistent state. On top of that, we can further tune the system performance.

There are several specific options if global snapshot is chosen to solve our consistency problem. In this article, we analyze some of the options concerning the degree of synchronicity. Namely, some approaches may interrupt the application itself severely while others may do it in a milder way. In the following sections, we talk about three approaches with different degrees of synchronicity. The first one is most synchronous, the second one is most asynchronous, and the third one is between them.

2.2 Totally Synchronous Global Snapshot

So how should we record the global state of a distributed system with correct consistency while its multiple nodes keep changing and there are in-flight messages in the channels? Easy - stop it. Of course, after the incoming data is blocked, the stream processing system will gradually drain up the channels and finally what we need to do is just to take each node's state and combine them.

But the drawback is also apparent. During the global snapshot, the whole application is blocked until the ongoing tuples are drained up, and the individual node states are collected.

This approach may not be suitable for production-level deployment but is useful for our comparison.

2.3 Totally Asynchronous Global Snapshot

We now move to a more realistic approach of global snapshot, which has the minimal side effect to the running application.

This is the famous Chandy-Lamport Algorithm[5]. Basically, in Chandy-Lamport, every node uses the received/sent marker messages to identify when to record the node's state and when to start/stop recording the application messages in the channels. The causality correctness is based on the assumption that the channels are FIFO. The markers fly from node to node, but they do not interfere with the regular application messages and do not stop the running of the whole application. The main time overhead is just some computation time for recording states and dealing with markers.

Chandy-Lamport is a very brilliant algorithm, especially regarding its delicate manipulation of markers to ensure correctness. However, compared with our previous clumsy approach, its space overhead is much higher. Because other than the nodes' states, we now need to store the messages in the channels as well. Also, the algorithm is much more complex.

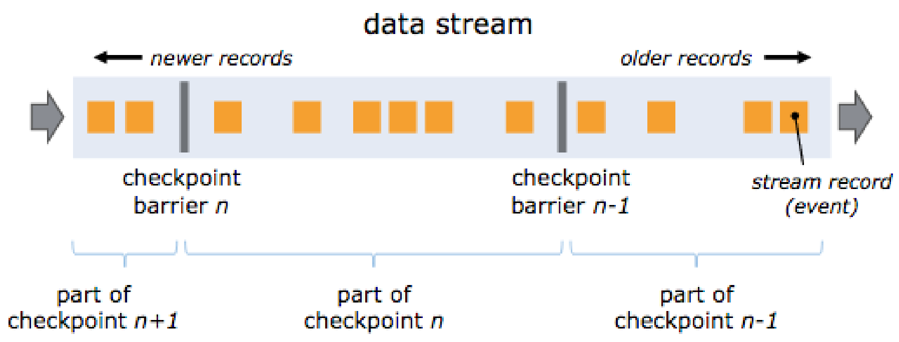


Figure 2.1: Data stream in Flink [6].

2.4 Partially Asynchronous Global Snapshot

Apache Flink offers a simple global snapshot approach that does not impact the performance of the normal application too much.

The authors of Flink describe Flink's approach as an "Asynchronous Barrier Snapshotting (ABS)" [7] one, and it is inspired by Chandy-Lamport. It has the same system assumption that the channels are FIFO. Just like Chandy-Lamport uses marker messages to collect the states belonging to a snapshot, Flink uses barriers. As illustrated in figure 2.1, barriers are special tuples with version IDs, which are inserted into the stream by the spouts as version indicators. We call the continuous tuples between adjacent barriers a *tuple block*. When an intermediate node has received all the barriers with the same version ID from all the input channels, it will checkpoint its state and relay the barrier to the next nodes. When a sink node has received all the barriers with the same version ID, it will acknowledge the checkpoint coordinator. If the checkpoint coordinator has received that same barrier from all the sinks, a snapshot is done. Also, the corresponding input data before that barrier will not be used any longer and can be deleted if necessary.

This method is straightforward, so does its name. It is based on barriers to work. It is called asynchronous because it does not block the whole application like the totally synchronous approach.

Nevertheless, it is not so asynchronous as Chandy-Lamport. The synchronization occurs at the node level. As figure 2.2 shows, the nodes with multiple input channels need to align the barriers from different input channels. If the barrier arrives earlier in an input channel, the upcoming application tuples from that channel must be buffered to avoid mixing tuples of different

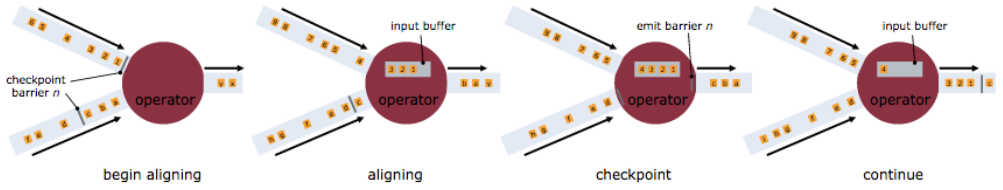


Figure 2.2: Align flows in Flink [6].

versions. Only when the same barrier arrives in the last channel, can the node record its state and relay the barrier. Before that, the execution of the node is partially blocked. This blocking and delaying may be even worse when there are more input channels, and the variance of the barrier arrivals is significant.

In reality, this kind of synchronization and entailed delay should not be deadly. As long as there is enough computing resource to consume the buffered tuples and there is enough space resource to buffer the tuples, it should be okay.

CHAPTER 3

DESIGN

3.1 Shortcoming of Flink

By the global snapshot approach of Flink's ABS, we can continuously dump the system's consistent state to a backup. But that is only one part of fault tolerance. Another part is how to recover from the backup in case of failures.

The recovery process is also intuitive. First, we need to find the latest global snapshot and recover each node with it. Then, the buffered input data (which was stored durably) after the barrier of that global snapshot needs to be replayed before new input data starts flowing. Eventually, the recovery goes seamlessly.

This whole solution of backup and recovery should be theoretically correct. However, by taking a closer look, we can find some performance issues.

- During backup, there is a freshness gap between the latest global snapshot and the current application state. A global snapshot is initiated when the spouts insert the corresponding barriers into the normal tuple stream, and that global snapshot can only complete when all the sinks have received that barrier from all their input channels. During this period, new tuples following that barrier keep coming and being operated by the nodes and the application state keeps changing, but the previous snapshot can not capture those changes.

- During recovery, there is a latency before the system can start processing the new incoming data. The buffered input that has not been acknowledged to be done by all the sinks should be replayed. This replaying delays the execution on new data.

- During recovery, the downstream nodes can not start working as soon as the recovery is requested. The tuple stream has to flow gradually from the upstream nodes to downstream nodes. When the buffered input tuples are replayed from the spouts, the upstream nodes can execute early, but the downstream nodes have to be idle until the replayed tuples reached them.

Although several drawbacks of Flink's ABS are discussed above, those drawbacks are somewhat mutual equivalent. They are just different prospectives to the same nature. The essential entity here to unify those prospectives and quantify the severity of the problem is the moving window that durably buffers the unacknowledged tuples - we call it *pending window*. The pending window is related to the freshness gap because its head indicates the latest snapshot and its rear indicates the latest arrived data. The pending window is related to the latency before new data can be processed because pending window should be replayed first. The pending window is related to nodes' starting time because the pending window is placed at the spouts. The size of the pending window can be roughly estimated by the number of new tuples generated by the spouts during the time for a barrier flowing through the critical path of the topology.

3.2 Modification Idea

To alleviate the problems mentioned above, we propose our *segment backup* method.

We already know that the pending window is essential to those problems, so our main idea is to modify the pending window. We change the single large pending window into multiple small ones positioned at some intermediate nodes as well the spouts. We call those intermediate nodes with associated pending window *connectors*. And the component between adjacent connectors (inclusively) is referred to as a *segment*. More precisely, a segment is a subtree originating from one connector, *segment-start connector*, and covers all its descendants until other connectors, *segment-end connectors* are reached and included as the leaves of this subtree.

Here is an easier way to think about the data stream. For each tuple block in the stream, when it exhausts a segment of the topology, we make the partial results (including node states and output tuples) durable at the segment-end connectors' pending windows and delete the corresponding tuples from the durable storage of the segment-start connectors' pending window. Segment by segment, the tuple blocks flow through the topology.

Our approach is based on Flink's ABS, so evidently, it shares a lot of similarities with ABS. Alike ABS, tuple blocks of the data stream are delimited by inserted barriers, and each node will checkpoint its state after receiving the same tuples from all the input channels. (And still, the faster channels may be blocked temporally to align with the slower channels.) The key differences are where the pending windows are and when to *truncate* (i.e., safely delete the pending tuples before a barrier) the pending windows. In ABS, only the spouts have pending windows and they will only be truncated to a version when all the sink nodes have received the corresponding barriers from all the incoming channels. But in segment backup, intermediate nodes also can have pending windows that can materialize the partial results. As a result, the spouts can truncate their pending windows as soon as a tuple block has flown through the first segment and the segment ending connectors have stored the output tuples durably (and checkpointed their node states, of course). Similarly, the following segments will continue this process until completion, in which the sink nodes may also want to store the final output tuples durably.

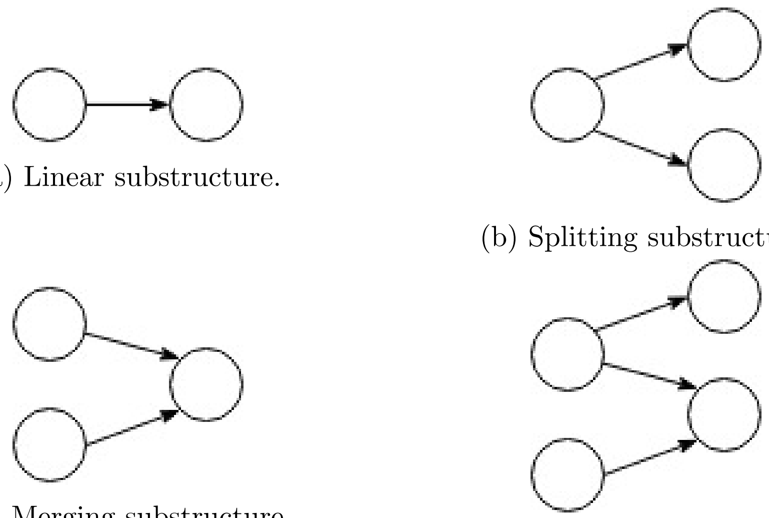
The benefits of having this modification can also be observed from different angles.

- The freshness gap of the backup and the current state is shortened. The materialization of the computation results happens as soon as the tuple block finishes a single segment.

- New tuples after a failure can be served earlier in the recovery process. Because the spouts' pending windows are truncated earlier during normal execution, the pending tuples at the spouts to be replayed during recovery are also fewer.

- Multiple nodes can resume working simultaneously in case of failures. When recovery happens, all the connectors can start replaying the tuples in their pending windows. Also, the idle time for all the normal nodes is shortened.

To sum up, the backup is fresher and/so the recovery is faster than ABS.



(a) Linear substructure.

(b) Splitting substructure.

(c) Merging substructure.

(d) Mixed substructure.

Figure 3.1: Four different substructures in a topology.

But what is the cost for this kind of improvement? It is imaginable that the time and space overhead increases during the normal execution without failures. Hence, we trade the performance of normal execution for the performance of the failure recovery. The more detailed discussion about the system trade-off is intentionally put off to chapter 4.

Another concern about our segment backup is whether it is still a global snapshot from which the system can be recovered correctly. After all, the pending window is split into multiple ones, and multiple connectors are directly responsible for recovery now. Is it still a consistent solution? Section 3.4 and section 3.5 will answer these questions by describing the details of the backup and recovery logic.

3.3 Substructures in a Topology

Before talking about the mechanism, let's first do some taxonomy on the substructures in our topology to clarify the arrangement of this chapter's remaining content.

Figure 3.1 lists four general substructures we can find in a DAG topology. Since the bolts are just dumb nodes without any significant duty in segment backup, the nodes shown in this figure are all connectors. The bolts are collapsed into the edges.

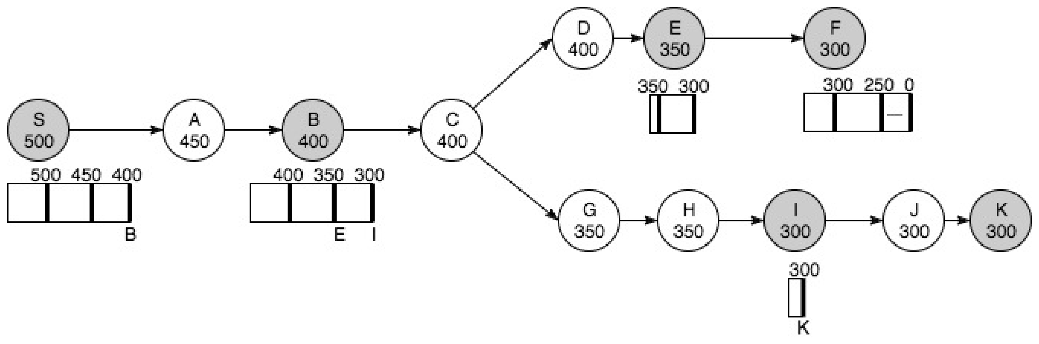


Figure 3.2: Example of segment backup.

(a) The linear substructure is the simplest one. Most of the real-life applications are just simple linear ones (i.e., pipelines). As expected, it should be eligible for any mechanism described later.

(b) The splitting substructure is a little more complex. Nonetheless, we can apply our solution on it naturally, because a segment is defined as a special subtree when it was incipiently introduced in 3.2.

(c) The merging substructure maps to operators like *join*. At first glance, it seems hard. But we will show in 3.6.1 that this case fits into our solution well.

(d) The mixed substructure is the most complicated. Still, we can apply our method directly. It will be shown in 3.6.2 that mixing splitting and merging substructures does not increase the difficulty.

If not specifically stated, the topology involved below does not have merging and mixed substructures.

3.4 Backup

Figure 3.2 shows an example state of a running stream processing system with segment backup enabled. The nodes in shadow are connectors, otherwise normal bolts. The number in each node means the version that the node has checkpointed. The connectors may have associated pending window which is shown under them. The only exception is for some sink nodes that do not care about the persistence of the final results. The tuples in each pending window

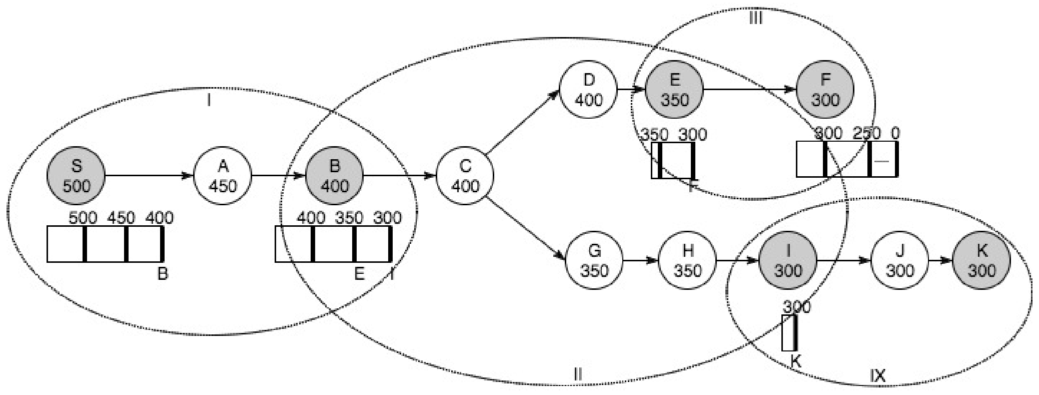


Figure 3.3: The connectors cut the topology into segments.

are those that have been produced by each connector but have not been acknowledged by the downstream connectors. The thick lines in the pending windows indicate the barriers. For simplicity, barriers' IDs are multiples of 50 in our example. The letters under some barriers are pointers that indicate the latest version acks sent by the corresponding downstream connectors. Those *version pointers* are especially useful when a connector has multiple downstream connectors.

Besides of those physical components, segment backup logically partitions the whole topology into segments. As figure 3.3 shows, there are four segments in our previous example, and each segment begins with one segment-start connector and finishes with the segment-end connector(s).

That is what each component stands for. Now, we move on to discuss what each component does for backup.

Every single node, no matter it is a connector or not, will checkpoint its state after receiving the same barriers from all the input channels, and in some cases temporarily block some faster channels to allow alignment of the barriers. For example, node A has version 450 checkpointed, which means it has at least finished processing all the tuples up to barrier 450.

Additionally, the connectors need to store all of their output tuples durably in their pending window. For example, node B has checkpointed its node state of version 400, and it has also pushed barrier 400 into its pending window. Note that output tuples are pushed into the pending window tuple by tuple, so newer output tuples after the barriers are pushed into the pending window continuously. Besides of keeping materializing its output tuples, a connector needs to send back version ack messages to all its adjacent upstream connectors. Once an upstream connector receives all the same version ack messages from its adjacent downstream connectors, it can truncate its pending window to that version safely. For example, node B has sent its upstream connector, node S, a version ack of 400, so node S has truncated its pending window to version 400 safely.

Because of materialization, the tail of a pending window is quite random and actively increasing. Because of truncation, the head of a pending window is always immediately after a barrier and only changes when a truncation happens. (To be more accurate, the head of a pending window should be k * 100 + x where k is a non-negative integer and x is randomly in (0, 50), because there may be a timestamp gap between the barrier and the next application tuple. For simplicity, we use barriers' ID to label the heads of pending windows.)

There are also some interesting observations from figure 3.3.

The pending window of node B has only been truncated to version 300. This truncation is because the downstream connectors of node B, which are node E and node I, have a largest common version ack of version 300. This relationship can be verified by the latest checkpoint of these two nodes. Even though node E has already sent back a version ack of version 350, its peer node, node I, processes slower and drags the next truncation at node B.

Different sink nodes may treat the output differently, but it does not affect the rest parts of the system. Node F may want to store the final output results; then it has a ”pending window” too. Node F will push its output tuples to this storage just like other connectors do, but this storage will not be truncated because there are no downstream connectors to acknowledge the data consumption. In contrast, node K may only care about displaying the results temporarily, so it does not need any pending window at all. This kind of connectors still need to acknowledge its upstream connectors when applicable but will default the pending window operations to no operation (NOP). It will only make a recovery easier, as we will discuss in the next section.

The lengths of the pending windows in the whole system can vary a lot. (The factors will be discussed deeply in chapter 4.) The segment from node S to node B has two tuple blocks pending; while the segment from node I to node K does not have even one tuple blocks pending, which is a very desirable condition.

3.5 Recovery

In case of failures, the backed up state of the nodes and the channels can be retrieved for recovery.

3.5.1 Recovery Logic: Replay from a Consistent Global State

The retrieved data should be used to construct a consistent restarting state before replaying. The essence is that the global snapshot used for recovery is not fixed as a whole one during a backup process, but is instead formed up flexibly after a failure occurs. The states of the segments are glued together by the pending windows so that each segment can recover to its freshest available states separately.

The core restarting logic goes on like the following procedure.

1. Each connector rolls back the node state to its *best safe version* (i.e., its latest checkpointed version).

2. Each connector *rewinds* (i.e., deletes the pending output tuples) the tail of its pending window to that best safe version to be coherent with its node state.

3. Each normal bolt rolls back its node state to the version of its closest upstream pending windows's head.

4. After each node has been restored and each pending window has been rewound correctly, the pending windows can start replaying the tuples. To avoid duplicate computation, each connector will ignore all the input tuples older that its version until any newer tuple arrives so that exactly-once semantics is ensured.

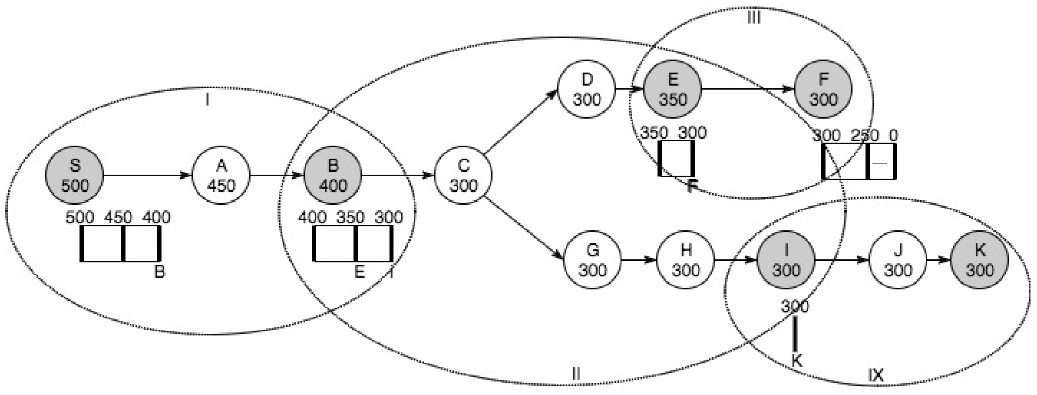


Figure 3.4: Example of segment recovery.

5. Finally, new tuples can start flowing into the system as soon as the spout has finished replaying its pending window.

During the whole process described above, the most important thing is the *adjusted state* prepared for replaying. The primary goal of that adjusted state is to construct a consistent global state that is as fresh as possible. The consistency of an adjusted state is described in the following two theorems.

Theorem 1. *In an adjusted state, the pending windows can be concatenated seamlessly along any spout-to-sink path of the topology with the help of version pointers.*

Theorem 2. *In an adjusted state, the nodes after a connector and ending with the next connector are in the same version.*

3.5.2 Recovery Example

To make it concrete, next we will show what will happen if some failure occurs when the stream processing system is exactly in the state of figure 3.3. Figure 3.4 illustrates the adjusted system state between a failure and a replay. The node states in this figure mean the current states for the nodes to restart with.

First, let's consider node S. Its best safe version of is 500, so it rolls back to version 500 and rewinds its pending window to that. Other nodes follow the same logic. Note that node E and node I roll back to different versions even though they are in the same segment. Node E does not need to sacrifice itself because of a slower peer.

When node B is replaying its pending window, node I will start processing the tuples as soon as it receives some. But node E will ignore the tuple block [300, 350] and only process tuples after version 350.

Node I has nothing to replay during recovery because its downstream path consumes its output efficiently. That is a desirable situation.

We can also verify theorem 1 and theorem 2 in our example. For theorem 1, the example shows that the pending windows of node S, B and I, which are [500, 400], [400, 300] and [300, 300], are directly adjacent hence form an integer. If we take node E's version pointer into account, the pending windows of node S, B, E and F, which are [500, 400], [400, 350] [350, 300] and [300, 0], are also consecutive.

3.6 Special Cases are Not Special

In our previous examples, only linear and splitting substructures are considered. In this section, we will also discuss the other two substructures.

3.6.1 Merging Substructure

So far, we have been talking about the single-input situation. But in reality, there may be multiple inputs in a topology. In this subsection, we will show that our previous recovery logic still works well for merging substructures. Without generality, an example about multiple spouts is discussed below.

Although it is ideal that different spouts can insert the barriers simultaneously, there may be some cases where different spouts can have different paces of adding barriers. To handle those cases, literally, nothing needs changing. Each spout can start its recovery process individually, and they will eventually converge.

In figure 3.5, there are two spouts, namely node S1 and node S2. Although node S1 has generated one more barrier than S2, the heads of their pending windows are the same version 400 because node C has sent the same version acks to those two spouts. Of course, node S1 needs to replay one more tuple block than node S2. But that only difference is absorbed by the spouts themselves automatically.

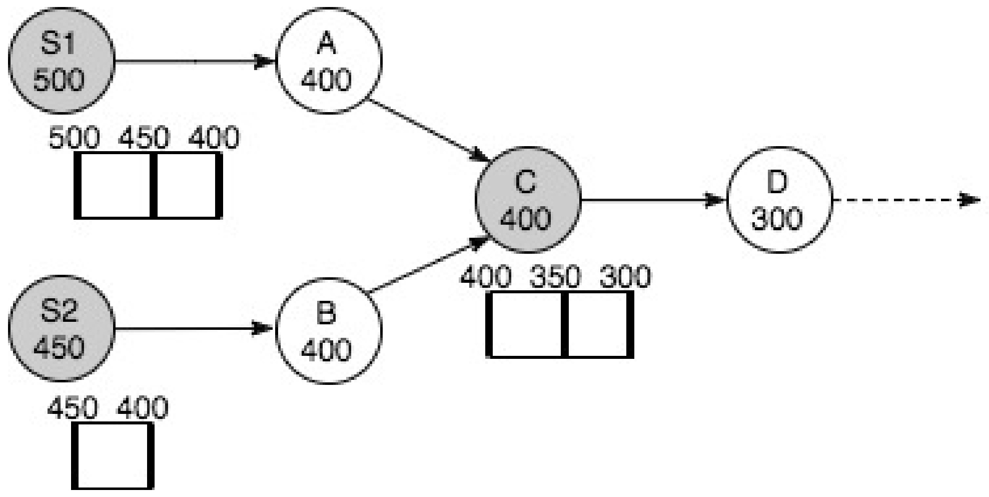


Figure 3.5: An adjusted state when multiple spouts exist.

3.6.2 Mixed Substructure

One step further from the previous example, we have a mixed substructure in figure 3.6. The difference is that a slower connector is in the downstream path of node S2. In this situation, node S2 needs to delay truncating its pending window during normal execution and replay a longer pending window. But those changes come with splitting substructure itself. The combination of splitting and merging substructure does not increase the complexity.

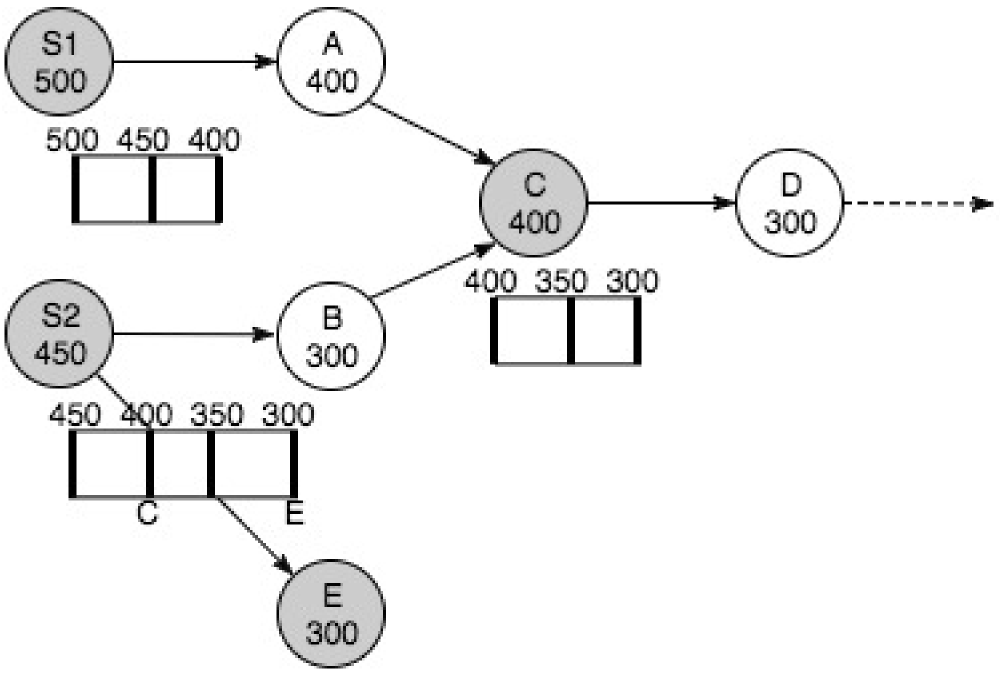


Figure 3.6: An adjusted state when mixed substructure exists.

CHAPTER 4

SYSTEM COST MODEL

It was mentioned in chapter 3 that we are faced with a trade-off of our design. Specifically, segment backup solution costs us additional time and overhead space for better recovery performance. In this chapter, we will try to optimize the cost by formulating this trade-off.

4.1 Optimization Problem

First of all, let's consider this problem in the whole picture and answer two fundamental questions in this framework.

4.1.1 Starting Point

We hope to speed up our recovery process, but still want to keep the additional overhead as low as possible. From this statement, we can formulate an abstract *constrained optimization* problem as following.

$$\begin{aligned} \underset{cfg}{\text{minimize}} \quad & \text{additional cost with configuration } cfg \\ \text{subject to} \quad & \text{decent recovery speed,} \\ & \text{other constraints.} \end{aligned} \tag{4.1}$$

From this basic starting point, we will add in the details step by step. Three aspects of quantification are discussed from section 4.2 to section 4.4.

4.1.2 What does the Cost Mean?

It has been mentioned many times in this article that the cost is the additional overhead during the normal running time of the stream processing system.

But we have two categories of cost herein, namely the time cost and the space cost. We only have one object function to optimize, so we need to unify those two kinds of cost. A natural and effective way to unify both the time and space costs is to weight them by their monetary prices. The accurate price can vary from machine to machine. In spite of that, we can use the accredited prices from cloud service providers (e.g., AWS, Azure, and GCP) to get an average sense.

4.1.3 What is the Optimization Variable?

The optimization variable in the problem statement 4.1 is named cfg. It should include the configuration introduced by segment backup. By reviewing the backup logic, we can see that there are two sets of variables.

1. We need to know whether an intermediate node is a connector or not. This configuration can be expressed as a bit vector $\boldsymbol{x}$ of length $|C|$ where C is the set of all the connectors.
2. We also need to know the frequency of barriers in the tuple stream. The spout decides this. Among l_1^{ob} (i.e., the length of spout's output tuple block) tuples that are generated by the spout, one is a barrier.

Thus, $cfg = \{\boldsymbol{x}, l_1^{ob}\}$.

4.2 Recovery Delay

The recovery delay is the time from when a recovery is requested to when it is completed. The request time is clear. And we define the completion of a recovery as following.

Definition 1 (Recovery Completion)**.** *The completion of a recovery is the moment when the last* [1] *node starts to operate on some input newer than what it has seen before the failure.*

For simplicity, we confine our discussion scope by some assumptions listed below.

[1]The meaning of "last" is temporal instead of spatial.

1. Only linear topologies are considered.
2. The completion of the recovery is approximated by the time when the last connector starts to process new tuples.
3. Each pending window has at least one tuple block so that it can start updating its descendant nodes without waiting for its upstream connector.
4. The sink node also needs persistence of its output tuples.
5. The backup is stored remotely, e.g., HDFS.

From definition 1, we can see that until the recovery finishes, the nodes are catching up to the failure point from the restored latest checkpoint. Hence, the expected number of tuples that each connector needs to reprocess is a half tuple block replayed by its upstream connector.

Using the parameters in table 4.1 and table 4.2, the recovery delay can be expressed as

$$t^r = t^{rr} + t^{rm} + \max_{i \in C \setminus \{\text{spout}\}} \left[(t^e + t^n) \times l_i^u + \max\left\{t^{fr}, t^{tr}, t^n\right\} \times \frac{l_{i-1}^{ob}}{2} \right], \quad (4.2)$$

where

$$\begin{aligned} l_{i-1}^{ob} &= l_1^{ob} \times r_{1,i-1}^o \\ &= l_1^{ob} \times \prod_{1 \le k \le i-2} r_{k,k+1}^o. \end{aligned} \quad (4.3)$$

The part $(t^e + t^n) \times l_i^u$ in equation 4.2 is the "propagation delay" for a single tuple; the part $\max\left\{t^{fr}, t^{tr}, t^n\right\}$ is the bottleneck processing delay for each tuple.

4.3 Overhead Time

We compare the overhead time cast by segment backup with the case where no fault tolerance is enabled. And we express the overhead time by computing the additional processing time for running the system for one second.

Table 4.1: Symbols about the System Delay

Symbol	Parameter	Comment
I/O		
t^e	edge delay	Average per-tuple network delay to next node.
t^{tr}	transmit delay	Average per-tuple transmit delay.
t^{fr}	read delay	Average per-tuple reading delay from a filesystem.
t^{fw}	write delay	Average per-tuple writing delay from a filesystem.
node		
t^t	node delay	Average per-tuple processing delay at a node.
t^n	node delay of normal tuple	Average per-normal-tuple processing delay at a node.
t^b	node delay of barrier	Average per-barrier processing delay at a node.
t^a	ack delay	Average per-ack processing delay (including both sender and receiver sides) at a connector.
operation		
t^{rr}	recover response delay	Average delay to issue a restart when failure occurs.
t^{rm}	recover misc delay	Average delay (e.g., adjusting states) before nodes are restarted.

Table 4.2: Symbols about the Application

Symbol	Parameters	Comment
topology		
N	node set	The IDs of all the nodes. $1, 2, ..., \|N\|$.
C	connector set	The IDs of all the connectors.
l_i^u	upstream length of connector i	The number of hops to reach node i's upstream connector.
output		
$r_{i,j}^o$	output ratio between node i and j	The number of output tuples of j over that of i.
l_i^{ob}	length of node i's output block	Average number of tuples in an output tuple block of node i.
rate		
f^o	original tuple frequency	The number of normal tuples generated by the spout per second.
f^t	tuple frequency	The total number of tuples generated by the spout per second.
λ_i	average arrival rate	Average arrival rate at node i's input buffer.
μ_i	average service rate	Average rate for a bolt to process a tuple.
size		
s^t	tuple size	Average tuple size in byte.
s^b	barrier size	Average tuple size in byte.
s^c	checkpoint size	Average checkpoint size of the node state in byte.
s^a	ack size	Average size of a version ack in byte.

The overhead time during normal execution exists at least in three forms.

1. Each node needs to handle the barriers, e.g., checkpoint the node state and relay the barriers. (Temporarily blocking some input channel should be included if merging substructure is also considered).
2. Each connector needs to do additional work to handle barriers, i.e., send version ack and deal with received version ack.
3. Each connector needs to materialize its output tuples.

Combining those three kinds of overhead time, we have the total time as

$$\begin{aligned} t^{oh} =& \frac{f^o}{l_1^{ob} - 1} \times t^b \times |N| \\ &+ \frac{f^o}{l_1^{ob} - 1} \times t^a \times (|C| - 1) \\ &+ \frac{f^o \times l_1^{ob}}{l_1^{ob} - 1} \times \sum_{i \in C} t^{fw} \times r_{1,i}^o. \end{aligned} \tag{4.4}$$

From the third term of equation 4.4, we can see that it is important to choose our connectors smartly. The nodes with smaller output ratio are preferred when other conditions are the same.

4.4 Overhead Space

The overhead space counts as communication and storage costs. We will discuss them separately.

4.4.1 Overhead Communication

The overhead communication is easy to calculate after we have talked about the overhead time. It is just the counterpart of the overhead time, namely, the network resource to send the barriers, to handle the version acks and to materialize the output.

The overhead communication per source tuple in byte is

$$
\begin{aligned}
n^{oh} =& \frac{f^o}{l_1^{ob} - 1} \times (s^b \times (|N| - 1) + s^c \times |N|) \\
&+ \frac{f^o}{l_1^{ob} - 1} \times s^a \times (|C| - 1) \\
&+ \frac{f^o \times l_1^{ob}}{l_1^{ob} - 1} \times \sum_{i \in C} s^t \times r_{1,i}^o.
\end{aligned} \tag{4.5}
$$

4.4.2 Overhead Storage

To compute the overhead storage, the principal work is to calculate the size of the pending windows. We will use M/M/1 queues [8] to model the system and apply Little's law [9][10] to find the size of the pending windows.

For future reference, we first introduce Little's law, which is helpful for estimating the number of customers in a system.

Lemma 3 (Little's law)**.** *The long-term average number of customers in a stable system L is equal to the long-term average effective arrival rate, λ, multiplied by the average time a customer spends in the system, W; or expressed algebraically: $L = \lambda W$.[11]*

We will use this general result later in two levels.

Now we move on to model our system using M/M/1 queue.

Definition 2 (M/M/1 queue)**.** *An M/M/1 queue represents the queue length in a system having a single server, where arrivals are determined by a Poisson process and job service times have an exponential distribution. [**?**]*

Our system complies with definition 2. The generation of source tuples by the spout is a Poisson process with rate $\lambda_2 = f^t$ [2]. Each bolt's processing delays have an exponential distribution with service rate $\mu_i = \frac{1}{t_i^t}$. Also, we have $\lambda_i < \mu_i$ to avoid buffer explosion.

The whole system is a pipeline of M/M/1 queues, as shown in figure 4.1. Each subsystem is an instance of M/M/1 queue. In contrast with the original model, this variant has two differences.

[2]To mock the reality better, the barriers should be inserted right after the normal tuples, and f^o should be used in code constantly. There will be some error in the distribution of inter-arrival time.

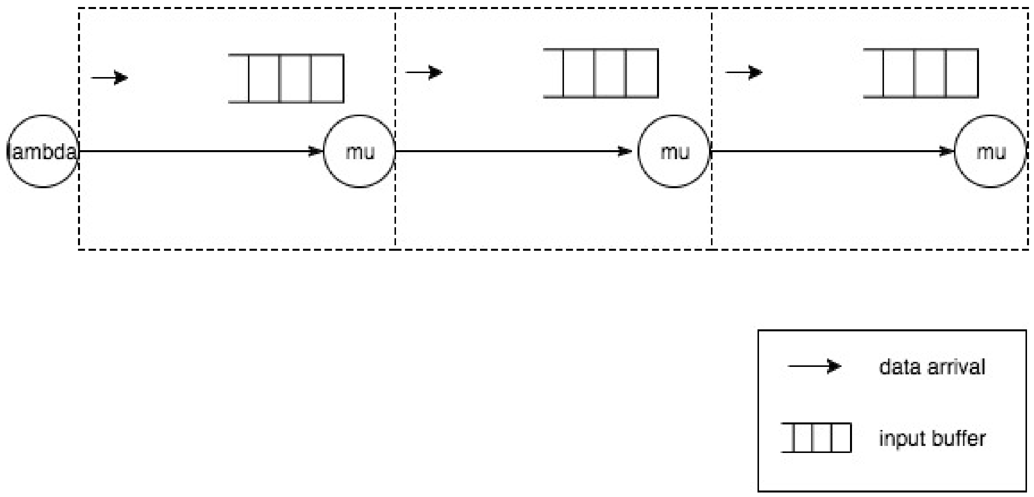


Figure 4.1: A pipeline of M/M/1 queues.

1. The data enters the subsystem when it is produced by the upstream node. If bandwidth is not a bottleneck in this subsystem, the data arrival event at the input buffer can be approximated by the data output event at the upstream node. At least, the arrival rate is equal.

2. Obviously, there are multiple queues instead of a single one now. In spite of that, each queue has an arrival rate proportional to λ_2. The first subsystem has the property of $\lambda_2 < \mu_2$. As a stable system, the output rate should be $r^o_{1,2}\lambda_2$. Iteratively, the arrival rate at node i's input buffer is $r^o_{1,i-1}\lambda_2$.

Thus, we can use the characteristics of M/M/1 queue on our pipeline variant.

By Little's law stated in lemma 3, we can derive the response time of an M/M/1 queue as indicated in theorem 4.

Theorem 4. *The average response time or sojourn time (total time a customer spends in the system) (of an M/M/1) does not depend on scheduling discipline and can be computed using Little's law as* $\frac{1}{\mu-\lambda}$*.[12]*

Applying theorem 4 directly in scope of each node, we have node i's response time as

$$
\begin{aligned}
t_i^s &= \frac{1}{\mu_i - \lambda_i} \\
&= \frac{1}{\frac{1}{t_i^t} - r_{1,i-1}^o \lambda_2} \\
&= \frac{1}{\frac{1}{t_i^t} - r_{1,i-1}^o f^t}.
\end{aligned}
\tag{4.6}
$$

Once again, we use Little's Law in the scope of each segment to calculate the size of pending windows. Connector i (except the sink node) has a pending window with the average length

$$
l_i^p = \lambda_{i+1} \times (\sum_{i<k\leq j} (t^e + t_k^s) + t^a), \tag{4.7}
$$

where j is the next connector.

And the total size of the pending windows is

$$
s_{total}^p = s^t \times \sum_{i \in C \setminus \{sink\}} l_i^p. \tag{4.8}
$$

The size of checkpoints at all the nodes is minimal compared to that of the pending windows. It is

$$
s_{total}^c = s^c \times |N|. \tag{4.9}
$$

After adding it to the total size of pending windows, we have the overhead storage

$$
s^{oh} = s_{total}^p + s_{total}^c. \tag{4.10}
$$

4.5 Total Cost

With all the system overhead calculated, we can estimate the monetary overhead per second by a weighted sum

$$
c = p^t \times t^{oh} + p^n \times n^{oh} + p^s \times s^{oh}. \tag{4.11}
$$

Table 4.3: Symbols about the Prices

Symbol	Parameters	Comment
p^t	computation time price	The price for computation per second.
p^n	network price	The price for sending one byte.
p^s	storage price	The price for storing a byte per second.

CHAPTER 5

EXPERIMENTS

We conduct experiments on our prototype implementation to validate the impacts of segment backup.

5.1 Prototype

To make our comparison with Flink effective, we implement a prototype of segment backup solution. The nodes are simulated as processes, and they communicate with each other by TCP. Our implementation is in Python. All the codes and tests are in GitHub [13].

5.2 Setup

The measurement uses linear topologies of seven nodes. But the number of connectors varies from two to four. No matter how many connectors there are, all the connectors are spread evenly along the path. Specially, the topology with two connectors (spout and sink) shows the basic idea of Flink. The topologies with intermediate connectors show the modification idea by segment backup. For simplicity, all the workload is all-pass filtering.

We run the tests on a computer with 2 GHz Intel Core i5 and 8 GB 1867 MHz LPDDR3. All the backup is stored on an HDFS set up on this computer via a virtual machine with 4 GB RAM.

For each topology, we run 50 tests to get the average of all the involved metrics. For each test, we run the application for 60 seconds, stop it, and restart it for another 20 seconds.

The average arrival rate at the spout is 30/s. The average service rate at each bolt is 80/s. The barrier ratio is 1/25.

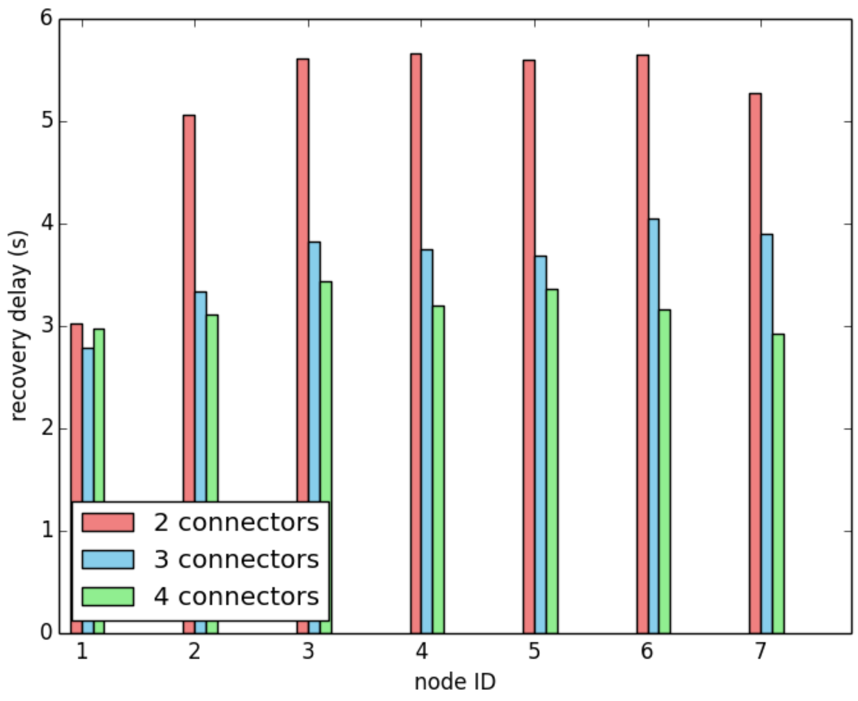


Figure 5.1: Per-node Recovery Delay.

5.3 Results

The experimental results are discussed in this section.

5.3.1 Recovery Delay

Figure 5.1 shows the per-node recovery latency during restart. The overall trend aligns with our expectation. When the number of the connectors increases, the recovery delay decreases because the intermediate results are materialized in time.

However, when the number of connectors increases from three to four, the recovery delay of the spout rebounds a little. This phenomenon may result from the simplicity of the spouts, which only need to replay their pending windows during a restart. Also, in our implementation, sending the pending tuples is non-blocking and only costs a short period. So the actions at the spouts are more predictable and stable. However, downstream nodes need to buffer all the replayed tuples and process them one by one, so longer pending

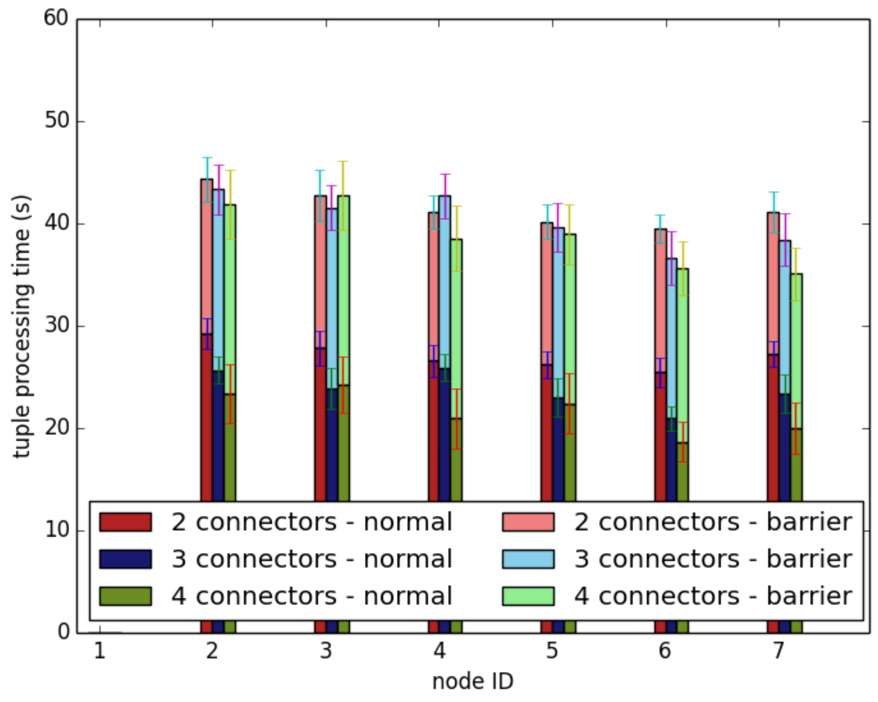


Figure 5.2: Per-node Processing Time (s).

windows will make their work more laborious.

5.3.2 Overhead Time

Figure 5.2 shows the per-node tuple processing time during normal execution. The dark bars stand for the time for normal tuples; the light bars stand for the time for barriers. Although the barrier ratio is only 1/25, the time spent on barriers is far more than 1/25 of that spent on normal tuples. That is because the bolts need to do many things while handling barriers. Each bolt needs to checkpoint the node states. Some bolts need to align the input channels. And each connector should finish a version of its pending window and deal with the version acks.

But this overhead time is not a big issue for two reasons.

1. In the real applications, the source tuples arrive at a much higher rate and the operation on them will be more complex than all-pass filtering. The side effect of the barriers will be weakened.

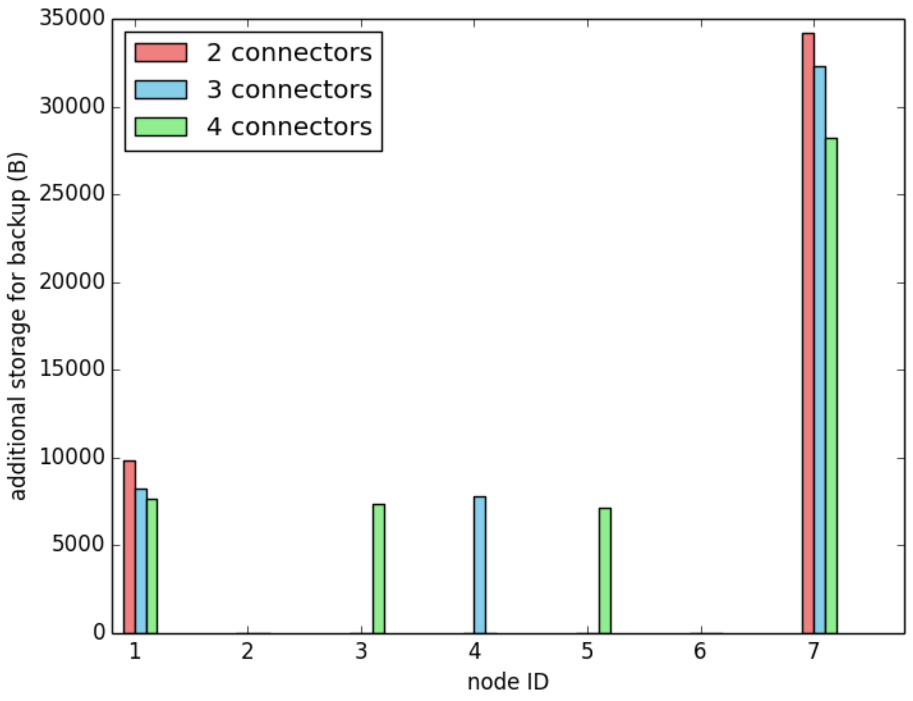


Figure 5.3: Per-node Additional Overhead Storage.

2. The overhead time for handling the barriers does not increase much with the number of the connectors. Most of this overhead is a nature of Flink's ABS; it is not imposed by segment backup.

3. The processing of version acks can have lower priority if the computation resource is limited. Temporarily delaying the sending of the version acks and the truncating of the pending windows will be fine, if the failure probability is low.

5.3.3 Overhead Storage

Figure 5.3 shows the per-node additional overhead storage. The overhead is measured as the backup size, most of which is the pending window. The decreasing trend at the spout is because the downstream connector that can truncate the spouts pending window is closer. The declining trend is not dramatic because there is some additional overhead time for dealing with the barriers. This correlation can be explained by Little's Law mentioned in

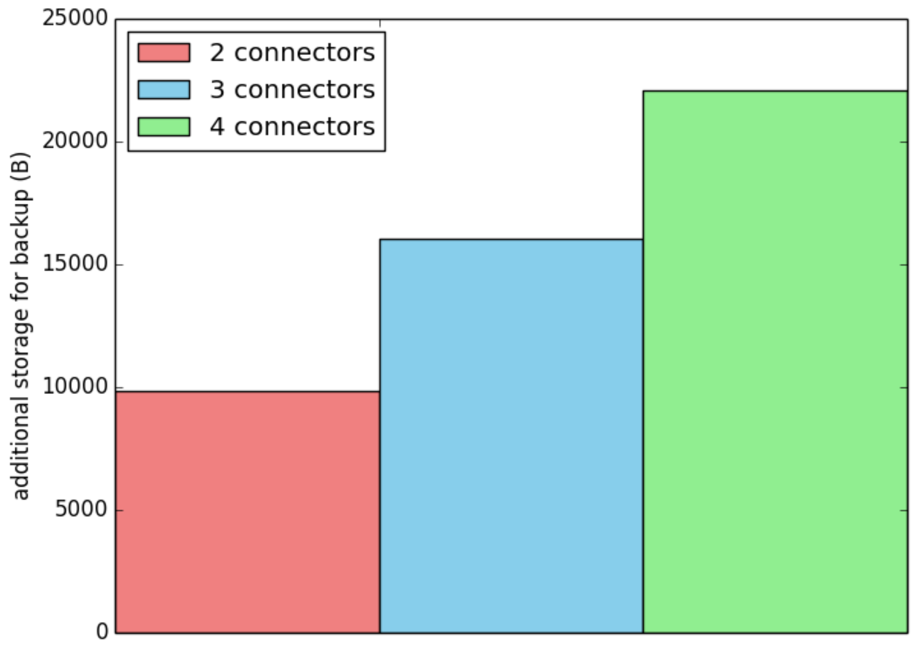


Figure 5.4: Total Additional Overhead Storage.

4. The number of the finished tuples at the sink node is decreasing. This also results from the additional overhead time.

Figure 5.4 shows the total additional overhead storage except the sink node. The total size of the pending windows increases when the number of the connectors increases. Again, this can be explained by Little's Law. The tuples under processing in the system are more because the service delay is increased.

Given the fact that storage price is becoming lower and lower, such overhead storage is acceptable.

CHAPTER 6

CONCLUSION

In this thesis, we presented segment backup which can tolerate datacenter disasters for stream processing systems, and we analyzed its cost model. Based on Flink's ABS, segment backup also uses barriers to decide backup versions. But segment backup adds intermediate connectors to materialize the partial results and constructs a global snapshot more flexibly in case of failures. Our experimental results show that increasing the number of connectors can shorten the recovery delay with acceptable overhead during normal execution.

REFERENCES

[1] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham et al., "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[2] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Data Engineering*, p. 28, 2015.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[4] "Apache samza," [Online; accessed 29-March-2017]. [Online]. Available: http://samza.apache.org/

[5] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 1, pp. 63–75, 1985.

[6] "Data streaming fault tolerance by apache flink," [Online; accessed 29-March-2017]. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-release-1.3/internals/stream_checkpointing.html

[7] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight asynchronous snapshots for distributed dataflows," *arXiv preprint arXiv:1506.08603*, 2015.

[8] L. Kleinrock, *Queueing systems, volume 2: Computer applications*. wiley New York, 1976, vol. 66.

[9] J. D. Little, "A proof for the queuing formula: L= λ w," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[10] J. D. Little, "Or forumlittle's law as viewed on its 50th anniversary," *Operations research*, vol. 59, no. 3, pp. 536–549, 2011.

[11] Wikipedia, "Little's law — wikipedia, the free encyclopedia," 2017, [Online; accessed 05-April-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Little%27s_law&oldid=775047024

[12] Wikipedia, "M/m/1 queue — wikipedia, the free encyclopedia," 2017, [Online; accessed 06-April-2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=M/M/1_queue&oldid=765962253

[13] J. Shen, "Segment backup," [Online; accessed 09-April-2017]. [Online]. Available: https://github.com/AspirinSJL/SegmentBackup