International Journal of Networked and Distributed Computing, Vol. 1, No. 4 (November 2013), 226-238

Fault Tolerant Distributed Stream Processing based on Backtracking

Qiming Chen
HP Labs, Hewlett Packard Co
Palo Alto, CA, USA
E-mail: qiming.chen@hp.com

Meichun Hsu
HP Labs, Hewlett Packard Co
Palo Alto, CA, USA
E-mail: meichun.hsu@hp.com

Castellanos Malu
HP Labs, Hewlett Packard Co
Palo Alto, CA, USA
E-mail: castellanos.malu@hp.com

Received 21 March 2013

Accepted 17 August 2013

Abstract

Since distributed stream analytics is treated as a kind of cloud service, there exists a pressing need for its reliability and fault-tolerance, to guarantee the streaming data tuples to be processed in the order of their generation in every dataflow path, with each tuple processed once and only once. Currently there exist two kind approaches: one treats the whole process as a single transaction, and therefore suffers from the loss of intermediate results during failures; the other relies on the receipt of acknowledgement (ACK) to decide whether moving forward to emit the next resulting tuple or resending the current one after timeout, on the per-tuple basis, thus incurs extremely high latency penalty. In contradistinction to the above, we propose the *backtrack* mechanism for failure recovery, which allows a task to process tuples continuously without waiting for ACKs and without resending tuples in the failure-free case, but to request (ASK) the source tasks to resend the missing tuples only when it is restored from a failure which is a rare case thus has limited impact on the overall performance.

The specific hard problem for building a transaction layer on-top of an existing stream processing platform consists in how to keep track the physical input/output messaging channels in order to realize re-messaging during failure recovery. Our solution is characterized by tracking physical messaging channels logically, for that we introduce the notions of *virtual channel, task alias* and *messageId-set* in reasoning, recording and communicating the channel information. We also provide a *designated messaging channel*, separated from the regular dataflow channel, for signaling ACK/ASK messages and for resending tuples, in order to avoid interrupting the regular order of data transfer.

We have implemented the proposed mechanisms on *Fontainebleau*, the distributed stream analytics infrastructure we developed on top of Storm. As a principle, we ensure all the transactional properties to be system supported and transparent to users. Our experience shows the novelty and efficiency of the proposed mechanisms.

Keywords: stream processing, fault tolerance, checkpointing, failure recovery.

Published by Atlantis Press

1. Introduction

1.1 *Transactional Stream Processing*

Stream analytics as a cloud service has become a new trend in supporting mission-critical, continuous dataflow applications. This has given rise to the reliability and fault-tolerance of distributed stream processing.

A stream is an unbounded sequence of events, or tuples. Logically a stream processing *operation* is a continuous operation applied to the input stream tuple by tuple, deriving a new output stream. In a distributed stream processing infrastructure, physically a logical ***operation*** may have multiple instances running in parallel, called ***tasks***. A graph-structured streaming process is a continuous dataflow process constructed with distributed tasks over multiple server nodes. A task runs cycle by cycle for transforming a stream to a new stream, where in each cycle it processes an input tuple, updates the execution state and emits the resulting tuples, carried in messages, to its target tasks.

The goal of transactional stream processing is to ensure that in every dataflow path the tuples are processed in the order of their generation, with each processed once and only once. With this goal, the failure recovery of a task must ensures the possibly missing input/output tuples to be re-acquired/re-sent but without violating the above "once and only once" semantics. It is generally based on checkpointing data processing state and messages for redoing.

1.2. *Prior Art*

The issue of failure recovery of distributed dataflow systems has been investigated in different contexts.

In the context of general distributed computing, an application involves multiple mutually dependent tasks [5,13] where the transaction semantics is based on the instant consistency of the global state. However, in a graph-structured streaming process, a task depends on its upstream neighbors only; and the states exposed to outside world are expressed as the output streams[1,8,9,11,12,15]. Therefore, instead of instant consistency of global state, the goal of transactional stream processing is to keep the **eventual consistency** [3]; this difference raises specific technical challenges.

In the context of database transactions, the previous work on transactional stream processing was mostly based on the notion of *snapshot isolation* [2,4,6,7,10] with the motivation to split a stream into a sequence of bounded chunks in order to apply the semantics of database transaction to each chunk, i.e. put the operation on a data chunk in a transaction boundary to yield a state snapshot. In this way, processing a sequence of data chunks generates a sequence of state snapshots. However, this mechanism only deals with the *state oriented* transaction boundary without addressing failure recovery.

Around the concepts of checkpoint and message logging there exist several approaches. One approach - the Storm's "transactional topology", treats the whole streaming process as a single operation which suffers from the loss of intermediate results in the occurrence of failures [16]. The other requires a task to wait for acknowledgement (ACK) to decide whether to move forward to emit the next tuple, or to resend the current one, on the per-tuple basis [14,15]; if the task has not received the ACK after timeout (e.g. between the target task fails and then heals) it will resend the output tuple, again and again, until being acknowledged. Although the "once and only once" semantics can be enforced by identifying the output tuples with their sequence numbers the in each dataflow channel and accordingly ignoring the duplicate ones, waiting for ACK and re-emitting output on the per tuple basis causes extremely high latency.

1.3. *Proposed Backtrack Recovery Mechanism*

To overcome the above drawbacks we developed an alternative methodology which allows a transactional task to process tuples continuously without waiting for ACKs and without resending output in the failure-free case; but ask its source task to resend the missing tuple only when it is restored from a failure. We refer to it as the ***backtrack*** or ***ASK-based*** recovery mechanism, for distinguishing it from the typical ***ACK-based*** recovery mechanism. Since failures are rare, backtrack the missing tuple would not have significant impact to the overall efficiency.

Based on this methodology we provided the algorithms for transactional *task execution* and *failure recovery*, and implemented them on top of Storm [16], the open-sourced distributed stream processing infrastructure.

1.4. *Implementation Problems and Solutions*

In order to build a transaction layer on top of an existing parallel and distributed stream processing system like Storm, rather than re-develop a new system from scratch, we have to solve some specific problems.

In a streaming process topology, a dataflow channel, or *messaging channel*, is identified by a pair of source and target tasks; a tuple is carried by a message and identified by the *message-id* composed with the channel and the corresponding sequence number. Supporting failure recovery based on checkpointing states and resending messages requires every task, when sending or receiving a message, to recognize the messaging channel; and specifically, requires a task to record the message-id before sending a message, in order to resend the right message to the right target task in case needed.

The problem is, however, common to the modern component based distributed dataflow infrastructures, the data routing between tasks is handled by separate system components inaccessible to individual tasks. For example, in a Map-Reduce platform, passing a resulted tuple from a Map task, M_{task}, to a Reduce task, R_{task}, is handled by the platform but unknown to M_{task} before emitting. More generally, with a distributed stream processing infrastructure such as Storm, when a task emits an output tuple, the destination depends on the grouping type, the current system state or the data content, which is unknown to the task thus cannot be record by it *before* emitting. As a result, the following *output channel paradox* may be caused.

- If a task T failed after it emitted an output tuple t to a target task T_1, when T is restored, it would re-emit the latest output tuple, t, anyway; however, under certain grouping criterion such as shuffle-grouping, the re-emitted tuple may go to a different target task, say T_2, since T_2 *never seen* t, it cannot determine whether t is duplicate and ignorable.
- If a task T failed and restored, the current input tuple may be missing, thus T would request each of its source tasks to resend its latest tuple; however, if a source task is unable to record its output channels *before emitting* every tuple, there is no way for it to know how to find the right tuple and resent it to the right target task.

Another hard problem in supporting message resending on top of an existing stream processing platform is the lack of accessible message re-sorting facility [14,15]. Let us consider the following situation, if a failed/restored task, T, has multiple source tasks, where the possible missing tuple came from is unknown to it (which we refer to as the *input channel paradox*); then T has to request all its source tasks to resend the corresponding latest tuples, with only one tuple being the one really missed. Although duplicate tuples can be ignored, resending those tuples through the ordinary dataflow channels may interrupt the order of regular tuple delivery, and the underlying infrastructure such as Storm does not support message re-sorting, or at least does not provide such APIs accessible to tasks.

In summary, building a transaction layer on top of an existing stream processing infrastructure imposes two *specific hard problems*: one is how to *keep track the physical input/output channels* for a task; another is how to *ensure resending tuples not to disrupt the regular order of data delivery*. In fact, these are common issues found in leveraging modern distributed dataflow systems thus worthy deep investigation.

Our solution to the output paradox mentioned above is to ***track the physical messaging channel logically*** by ***reasoning***; for that we introduce the notions of ***virtual channel***, ***task alias*** and ***messageId-set***, and use them in reasoning, tracking and communicating the channel information logically.

Our solution to keeping the regular order of data delivery during resending messages is to provide a ***designated messaging channel*** that is separated from the regular dataflow channel, for signaling ACK/ASK and resending tuples. With this additional messaging channel and the corresponding algorithm, the interruption to the regular order of tuple delivery during failure recovery can be avoided effectively.

We have implemented the proposed mechanisms on *Fontainebleau*, the distributed stream analytics infrastructure we developed on top of Storm [16]. As a principle, we ensure all the transactional properties to be system supported and transparent to users. Our experience shows that the novel virtual channel mechanism allows us to handle failure recovery correctly in elastic streaming processes, and the ASK-based recovery mechanism significantly outperforms the ACK-based one.

The rest of this paper is organized as follows: Section 2 outlines the concept of graph-structured distributed streaming process; Section 3 describes the backtrack-based failure recovery; Section 4 discusses how to track

messaging channels intelligently; Section 5 illustrates the experiment results; Section 6 concludes.

2. Distributed Streaming Process

Real-time stream analytics has increasingly gained popularity since enterprises need to capture and update business information just-in-time, analyze continuously generated "moving data" from sensors, mobile devices, social media of all types, and gain live business intelligence.

We have developed a massively *parallel, distributed* and *elastic* stream analytics platform, with code name ***Fontainebleau***, for executing *continuous, real-time* streaming processes. Our platform is built on top of Storm – a modern data stream processing system.

Architecturally *Fontainebleau* is characterized by the concept of **open-station**. In stream processing, data flow through *stationed* operators. Although the operators are defined with application logic, many of them have common execution patterns in I/O, blocking, data grouping, etc, as well as common functionalities such as the transactional control to be discussed in this report, which can be considered as their "meta-properties" and categorized for providing unified system support. This treatment allows us to ensure the operational semantics, to optimize the execution, as well as to ease user's effort for dealing with these properties manually which can lead to fragile code, disappointing performance and incorrect results. With the above motivation we introduce the notion of open-station as the container of a stream operator, and provide an open-executor with related system utilities for each open station. In the OO programming context, the open-executor is coded by invoking certain abstract functions (methods) to be implemented by users based on their application logic. We support transactional stream processing based on the open-station architecture, where a stream processing operator, *O*, is wrapped by a "transaction station", *S*, that provides designated system support for checkpointing, failure recovery, etc, but *open* for the application logic to be plugged-in. Accordingly, a task of *O* can be viewed as the instance of station *S*.

Let us observe a streaming process example for matrix data manipulation and analysis. In this streaming process, the source tuples are streamed out from "matrix spout" with each contains 3 equal-sized float matrices generated randomly in size and content (the application background is the sensor readings from oil wells). The tuples first flow to operation "`tr`" for transformation, and then to operation "`gemm`" (general matrix multiplication) and "`blas`" (a basic linear algebra operation) with "fields-grouping" on different hash keys; the output of "`gemm`" is delivered to operation "`ana`" (analysis) with "all-groupuing", and the output of "`blas`" to operation "`agg`" (aggregation) with "direct-grouping". The logical dataflow of this process is illustrated in Fig 1.

The partial specification of the graph structure, or topology, of this streaming process is listed below.

```
BlueTopologyBuilder builder = new BlueTopologyBuilder();
builder.setSpout("matrix_spout", matrix_spout, 1);
builder.setBolt("tr", tr, 2).allGrouping("matrix_spout");
builder.setBolt("gemm", gemm, 2).fieldsGrouping("tr", new
   Fields("site", "seg"));
builder.setBolt("ana", ana, 2).allGrouping("gemm");
builder.setBolt("blas", blas, 2).fieldsGrouping("tr", new
   Fields("site"));
builder.setBolt("agg", agg, 2).directGrouping("blas");
```

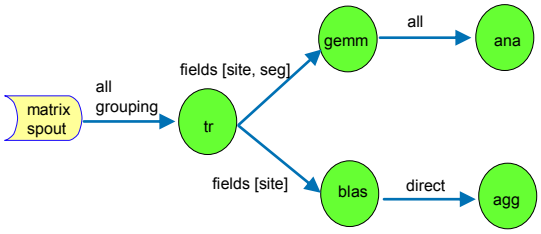

Fig. 1: A logical streaming process with operations, links and dataflow grouping types

Physically, each operation has more than one task instances. Given a pair of source and target operations, the tuples sent from the tasks of the source operation to the tasks of the target operation are grouped with various criteria, as illustrated in Fig 2.

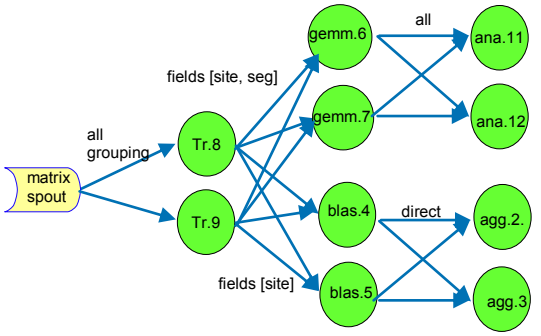

Fig. 2: A physical streaming process with each operation having multiple instance tasks

To identify the messaging channels and tuples we introduce the following notations.

- A topology-wise unique *task#* is assigned to each task by the underlying infrastructure.
- A *taskId* is the composition of the *task#* and the *operationId* (name) for that the task is an instance, denoted by *operationId.task#*; for instance, *taskId* "`agg.2`" is a task of operation named "`agg`".
- A message *channel* is identified by the source and target *taskIds*, denoted by *srcTaskId^targetTaskId*; for instance, a message channel from task `tr.8` to `gemm.6` is expressed as `tr.8^gemm.6`.
- A *messageId*, or *mid*, is identified by the channel and the message sequence number, say *seq#*, via this channel, as expressed by *channel-seq#,* or more exactly by *srcTaskId^targetTaskId-seq#*; for instance, "`tr.8^gemm.6-134`" identifies the 134th tuple sent via the channel from "`tr.8`" to "`gemm.6`".

A tuple transmitted through a messaging channel is identified by the *messge-id* (or *mid*).

Given a task, all the possible input and output channels can be extracted from the streaming process topology statically, but the input/output *mids* are resolved dynamically during execution since the exact physical channel may be data dependent, e.g. depending on the hash value of certain fields of a tuple.

As mentioned above, to build a reliable transaction layer on top of an existing parallel and distributed stream processing infrastructure like Storm, rather than re-develop a new system, we need to *track the physical dataflow channel logically* by *reasoning*, for that we will introduce the notions of ***virtual channel***, ***task alias*** and ***mid-set***, and use them in reasoning, tracking and communicating the channel information.

3. Backtrack Based Failure Recovery

We treat failure recoverability as a kind of task execution pattern, and provide the canonical open station framework to support it automatically and systematically.

3.1. *Architecture Overview*

Before going to details, let us first overview the task execution pattern of our platform. The stream processing operators are wrapped by station classes; and a transactional station is defined as an abstract class *BasicCkStation* with system supported semantics; however, as shown in Fig 3, it is open for users to plug-in their application logic by implementing the defined abstract functions.

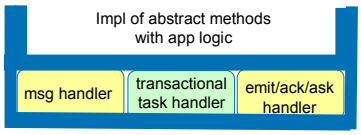

Fig. 3: BasicCkStation provides system support to transactional streaming task and open for app logic to plug-in

The *BasicCkStation* class provides three major system methods (as well as the supporting methods for them):

- prepare(Config, TopologyContext, OutputCollector) this is used to setup the initial system and application state for the task. This method is invoked when the task is initially setup or restored from a failure, therefore the recovery() method discussed below is invoked, if necessary, inside the prepare() method.
- execute(Tuple) that covers the per-tuple processing, checkpointing state, acknowledging source task, emitting outputs, etc.
- recovery(byte[]) which implements the recovery steps: rollback to the last checkpoint state from the serialized byte-array retrieved from disk or database; re-emit output to the right target tasks at the downstream, and re-acquire the latest input from the source task at the upstream. As explained in more detail later, all the source tasks will be contacted to ensure the missing tuple to be resent; resending tuples is via a specific messaging channel separated from the regular dataflow channel between tasks; and additional mechanisms are provided for a task to ignore duplicate inputs.
- Fields outputFields() that is used to specify the fields of the output tuple.

These methods invoke certain abstract methods which will be implemented based on application specific semantics (e.g. how to process each tuple) and resource specific properties (e.g. the specific checkpoint engine based on files or databases).

The *BasicCkStation* class provides two major application oriented abstract methods,

- void initAppState() that is used to specify the initial state and variables to be carried on for process each input tuple;

- List compute(Tuple) that is used to update task state, and to derive zero, one or more output tuples from each input tuple.

The structure of the *BasicCkStation* class is illustrated below.

```
public abstract class BasicCkStation extends BasicStation  {
   public void prepare(Config, Context, OutputCollector) { … };
   public void execute(Tuple) { … };
   public void recovery(byte[]) { … };
   ….
   public abstract ArrayList<Values> compute(Tuple tuple);
   public abstract void initAppState();
   public abstract Fields outputFields();
   ….
}
```

In order to create a transactional operation (tasks are instances of operations), the user only needs to define a corresponding class that extends the abstract BasicCkStation class, and implement the above abstract methods (and any inherited abstract methods), as shown by the following example.

```
public class GemmCkStation extends BasicCkStation {
   float alpha, beta;
   public GemmCkStation(float alpha, float beta) {
      super();   this.alpha = alpha;   this.beta = beta;
   }
   @Override
   public Fields outputFields() {
      return new Fields("site", "seg", "matrixC");
   }
   @Override
   public ArrayList<Values> compute(Tuple tuple) {
     ...
   }
   @Override
   public void initAppState() {
       ...
   }
};
```

With the open station architecture, the checkpointing and failure recovery are completely transparent to users as they only need to care about how to process each tuple for their applications.

3.2. *General Algorithms*

A task is a continuous execution instance of an operation wrapped by a station where two major methods are provided, one is the *prepare*() method that runs initially before processing input tuples; the other is *execute*() for processing an input tuple in the main stream processing loop. Failure recovery is handled *in prepare*() since after a failed task is restored it will experience the *prepare*() phase first.

Task Execution. A task runs cycle by cycle continuously for processing inputs tuple by tuple. The tuples transmitted via a dataflow channel are sequenced and identified by the seq#, and guaranteed to be processed in order; a received tuple, *t*, with seq# earlier than the expected will be ignored; later than the expected will trigger the resending of the missing ones to be processed before *t*. This ensures each tuple to be processed once and only once and in the right order. Further the state and data processing results on each tuple are checkpointed (serialized and persisted to file) for failure recovery. After checkpointing the transaction is committed, acknowledged and then the results are emitted.

For efficiency, a task does not rely on the receipt of "ACK" to move forward; instead, acknowledging is asynchronous to task executing and only used to remove the buffered tuples already processed by the target tasks. Since an ACK triggers the removal of the acknowledged tuple and all the tuple prior to that tuple, the ACK is allowed to be lost.

The algorithm of ***execution()*** is illustrated in Fig 4. The follows are worth noting in more detail.

- The seq# of each input tuple, *t*, is checked; if *t* is duplicate (with smaller seq# than expected) it will not be processed again but will be acknowledged to allow the sender to remove *t* and the ones earlier than *t*; if *t* is "jumped" (with seq# larger than expected), the missing tuples between the expected one and *t* will be requested, resent and processed first before moving to *t*.
- The checkpointed state consists of a list of objects; when check-in, the list is serialized into a byte-array to write to a binary file; when checkout, the byte-array obtained from reading the file is de-serialized to the list of objects representing the state.

The input/output channels and seq# are recorded before checkpointing and emitting. Since each output tuple is emitted only once but possibly distributed to multiple

destinations which are unknown to the task before emitting, the output channels must be "reasoned", recorded and checkpointed, to be used in the possible failure-recovery, which is a core investigation of this work and to be discussed later.

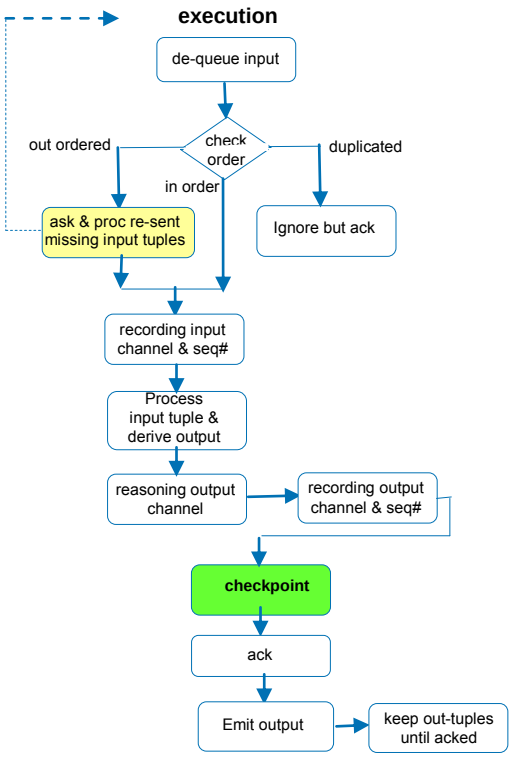

Fig. 4: Flowchart of task execution

Task Recovery. Supported by the underlying Storm platform, we have the failed task instance re-initiated on an available machine node by loading the serialized operation class to the node and creating an instance over there. As shown in Fig 5, recovery a failed task is a triple-folds problem:

- restore its execution state from checkpoint,
- request the possible missing input, and
- re-emit the last output.

When the failed and then restored task has multiple source tasks, it cannot determine where the missing tuple came from, therefore it has to ask each source task to resend the possible next tuple wrt the latest tuple it received from each input channel and recorded in its input-map that is a part of the checkpoint content. It in turn requires every pair of source and target tasks to have a protocol on identifying the "latest tuple", and this is why a task need to identify the physical dataflow channel and record the input/output seq#, before an output is emitted (i.e. before the output is possibly missing). The algorithm of *prepare*() is illustrated in Fig 5.

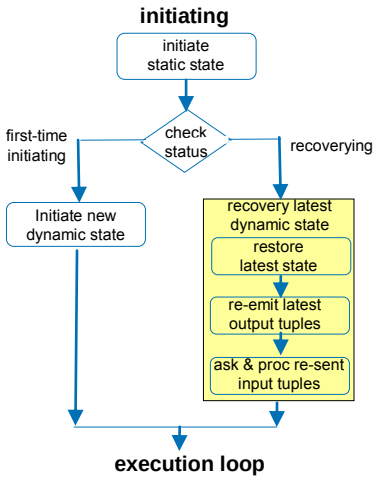

Fig. 5: Flowchart of task initiation/recovery

Second Messaging Channel. Another challenge in dealing with backtrack recovery is how to ensure the order of regular tuple delivery not interrupted by the task recovery process. Because the recovered task with multiple source tasks may receive more than one resent tuples, and besides the one really missing, the others, may have been delivered and queued but not yet taken by the task; in that case, appending the resent tuple to the queue would interrupt the order of the queued tuples. We solve this problem in the following way.

- A second massaging channel, separated from the regular dataflow channel, is provided for a task, for signaling ACK/ASK and resending tuples (Fig 6).
- When a task *T* is restored from a failure, it first requests and processes the resent tuples from all input channels, before going to the normal execution loop. In this way, if a resent tuple has been put in the input queue of *T* previously but not yet taken by *T*, that tuple can be identified as duplicate one and ignored in the normal execution loop.

For this designated messaging channel each task has a distinguish socket address (SA) and an address-book of its source and target tasks; the SA is carried with its output tuples for the recipient task to ACK/ASK through that messaging channel. Due to the change of SA when a task is restored from a failure (in that case the task may

even be launched to another machine node), and due to the unavailability of the SA in the first correspondence, a Home Locator Registry (HLR) service is provided.

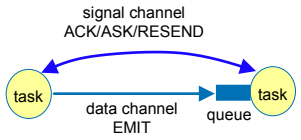

Fig. 6: The secondary messaging channel for ACK/ASK and resending tuples

4. Tracking Messaging Channels

As mentioned early, in the execution of a streaming process, each tuple is uniquely identified by a ***messageId***, or ***mid***. A *mid* is represented as *srcTaskId^targetTaskId-seq#*. For example, the *mid* "`tr. 8^gemm.6-134`" identifies the *134th* tuple sent from task "*tr.8*" to task "*gemm.6*".

Re-emit an output tuple or request/resend a missing input tuple during recovery, requires a task to comply with the peer tasks on the *mid* of the tuple. When a task T_1 sends a tuple to task T_2 through the messaging channel between them, T_1 must record the seq# via that channel ***before emitting*** the tuple, and T_2 must record the *seq#* upon receipt. This is trivial if the message routing is responsible by the underlying infrastructure and the sender may or may not know the exact destination before emitting. In fact this is the common situation in the component-based distributed computing infrastructure.

This situation has motivated us to track messaging channels *logically*. The general idea is for the sender task to express and record a *mid* in such a logical form that allows the recipient task to recognize the matched logical channel and physical channel, as well as allows the sender task to find the right tuple and resend it to the right recipient based on the "logical message identifier", in handling acknowledgements and in responding to re-send requests.

4.1. *MessageId-Set and Virtual MessageId*

We consider two kinds of "logical message identifiers", one related to a set of recipients, another related to a virtual recipient.

When an emitted tuple is delivered to multiple recipients through multiple message channels, we allow the tuple to be identified by a ***mid-set***. A *mid-set* contains multiple individual *mids* with the same source task but with different target tasks. On each recipient side, the target task picks up from that *mid-set* the *mid* with target *taskId* matching itself, for recording the input channel and seq# accordingly. The matched *mid* will be used for identifying both ACK and ASK messages. In the other words, *mid-sets* only appear in the sender task and recorded for output tuples; in the recipient side, only the matched single *mid* is recorded and used. On the sender side, to find the kept tuple that matches the *mid* carried by an ACK or ASK message is simply based on the set-membership relationship. As mentioned above, the tuple matches an ACK message will be garbage-collected, and the tuple matches an ASK message will be resent during failure recovery. A resent tuple is always identified by a single, matched *mid.*

Further we introduce the notions of ***task alias*** and ***virtual mid*** to resolve the destination of message sending with "fields-grouping", (or hash partition). In this case an output tuple only goes to one instance task of the given target operation which is determined by the routing component based on a unique number yield from the hash and modulo functions. Although the sender task has no knowledge about the physical destination before emitting a tuple, it can calculate that number, and can treat that number as the ***alias*** of the corresponding target task ID, then create a ***virtual mid*** using that *alias*. A *virtual mid* is directly recorded and used in both the source task that sends the tuple, and the target task that receives the tuple.

Below we illustrate how to use these notions to resolve the messaging channels wrt the typical grouping types.

All-Grouping. With "all-grouping", a tuple emitted by a task, e.g. `gemm.6`, is distributed to all tasks of the recipient operation (e.g. `ana.11, ana.12`). Since there is only one emitted tuple but multiple physical output channels, we use ***mid-set*** to identify the emitted tuple. For instance, a tuple sent from `gemm.6` to `ana.11` and `ana.12` is identified by

`{gemm.6^ana.11-96, gemm.6^ana.12-96}`

On the sender site (e.g. `gemm.6`), this *mid-set* is recorded and checkpointed; in each recipient task (e.g. `ana.11`) only the ***single mid*** matching itself (e.g. `gemm.6^ana.11-96`) will be extracted, recorded and used in ACK and in ASK messages. In the sender task (e.g. `gemm.6`) the match of the ACK or ASK message identified by a single *mid* and the recorded tuple

identified by a *mid-set* is determined by set membership. For example, the ACK or ASK message with *mid* `gemm.6^ana.11-96` or `gemm.6^ana.12-96` *matches* the tuple identified by {`gemm.6^ana.11-96, gemm.6^ana.12-96`}.

Fields-Grouping. With "fields-grouping", the tuples output from the source task are hash-partitioned to multiple target tasks, with one tuple going to one destination task; this situation is similar to have Map results distributed to Reduce nodes. With the underlying streaming platform (common to most other platforms), the target task ID is mapped from the hash partition index, ***a***, calculated based on the selected key fields list, *keyList*, over the number of *k* tasks of the target operation, as

a* = *keyList.hashcode() % k

On the source task, although it is impossible to figure out the physical target task and record the physical *mid* before emitting a tuple, it is possible to compute the above hash partition index, which allows us to use it as the ***task alias*** for identifying the target task. A task alias is denoted by

`operationName.`***a***`@`

such as `gemm.1@`, where ***a*** is the hash partition index.

In more detail, the output tuples of task "`tr.9`" to tasks "`gemm.6`" and "`gemm.7`" are under "fields-grouping" with 2 hash-partitioned index values 0 and 1, these values, 0 and 1, serve as the aliases of the recipient tasks. Then the target tasks "`gemm.6`" and "`gemm.7`" can be represented by aliases "`gemm.0@`" and "`gemm.1@`" without ambiguity. Although the task alias (`gemm.1@`) is different from the real target *taskId* (`gemm.6`), it is unique and all tuples sent to `gemm.6` will bear the same *target task alias* under the given field-grouping.

Then an output tuple, say, from task `tr.9` to `gemm.6` under "fields-grouping" is identified by the ***virtual mid***

`tr.9^gemm.1@-35`

where the target taskId `gemm.7` is replaced by the task alias "`gemm.1@`".

A *virtual mid*, such as `tr.9^gemm.1@-2`, is directly recorded at both source and target tasks and used in both ACK and ASK messages. There is no need to resolve the mapping between a task-alias and the actual task-Id represented by the alias.

In case an operation has two or more target operations, such as in the above example, the operation "`tr`" has 2 target operations, "*gemm*" and "*blast*", an output tuple can be identified by a *mid-set* containing *virtual-mids*; for instance, an output tuple from task "`tr.9`" is identified by the following *mid-set*

{`tr.9^blas.0@-30, tr.9^gemm.1@-35`}

that expresses that the tuple is the 30th tuple sent from "`tr.9`" to one of the *blas* task, and the 35th to one of the *gemm* task. The recipient task with the recorded alias `blas.0@`, can extract the matched *virtual-mid* `tr.9^blas.0@-30` based on the *match of operation name* "`blas`", for recording the seq# `30` for that virtual channel.

Global-Grouping. With global-grouping, tuples emitted from a source task are routed to the same instance task of the target operation; and the selection of the recipient task is made by a separate routing component outside of the source task. Our goal is for the source task to record the messaging channel before each tuple is emitted. For this purpose we do not need to know what the exact task is, but create a single *alias* to represent the recipient task. In this case, all tuples go to the same recipient task that is represented by the same alias; the latest seq# is recoded on both the sender and receiving sides.

Direct-Grouping. With direct grouping, a tuple is emitted using the emitDirect API with the physical *taskId* (more exactly, task#) as one of the parameter. For channel specific recovery we extend the Topology Builder to turn the rest of the grouping types not discussed above, to direct grouping; for each emitted tuple, the destination task is selected on-the-fly based on load balancing, i.e. the one currently with least load (i.e. least seq#) is chosen.

Shuffle-Grouping. Shuffle grouping is a popular grouping type. As mentioned above it is converted to direct grouping where a tuple is emitted to a designated task selected based on load balancing, i.e. the channel with least seq# is selected.

In summary, we track and record the message channels wrt various grouping types: for "all-grouping" the concept of *mid-set* is adopted; for "fields-grouping", *task-alias* and *virtual-mid* are used. We support "direct-grouping" systematically (rather than letting user to decide) based on load-balancing. Further we convert all other grouping types, which are random by nature, to our

system-supported direct grouping. The combination of *mid-set* and *virtual mid* allows us to track the messaging channels of a task with multiple grouping criteria.

4.2. *System Support for Channel Tracking*

For guiding channel resolution, we extract the topology information from the streaming process definition, and create the task specific meta-data objects: Task-Input-Context, ***TIC***, and Task-Output-Context, ***TOC***, for specifying input and output channels, grouping types, etc. Multiple *TIC* and *TOC* objects are associated with a task.

A task, *T*, has a list of TIC objects; with each specifying the input context of one source task of *T* ; it comprises the following:

- task ID of source task T_s, that is the key field of TIC;
- operation ID (name) of source operation O_s, of that T_s is an instance;
- grouping type (shuffle, field, … etc);
- channel;
- stream ID (abstract dataflow between the source operation O_s, and the operation of this task);

A task, *T*, has a list of TOC objects; with each specifying the output context of one target operation (with one or more target task instances) of *T*; it comprises the following:

- operation ID (name) of target operation, O_t, that is the key field of TOC;
- grouping type (shuffle, field, … etc);
- key indices (int []) indicating the key fields of output tuples for hash partitioning in the field-grouping case;
- channel list comprising the channels from this task to all the tasks of the target operation, O_t.
- stream ID (abstract dataflow between the operation of this task and the target operation O_t.

While the *TIC list* and the *TOC list* provide static grouping information, the actual input and output <channel, seq#> are recorded in the HashMaps, inChannelBook and outChannelBook, of each task. Note that the seq# is the latest (largest) sequence number.

Tracking Output Channel in Sending Task. A single tuple emitted from a task may go to one or more target tasks. Using *TOC*, these target messaging channels can be traced operation by operation. The messaging channels and seq#s are represented with either actual or virtual, either single or set, *mids*, and recorded in the outChannelBook of the task.

Fig 7 shows an example where operation OP_0 has two target operations having 3 tasks and 2 tasks respectively, and with "all-grouping" and "fields-grouping' respectively. For a task of OP_0, its TOC, is illustrated in Fig 8.

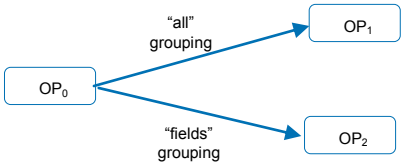

Fig. 7: A grouping example

Reasoning with this TOC, each output tuple from a task of OP_0 will be distributed to 4 target tasks, including 3 task instances of OP_1 (with all-grouping) and 1 of OP_2(with field-grouping); they are identified by a *mid-set* with 4 *mids*, with one of them (the one associated with field-grouping) is virtual.

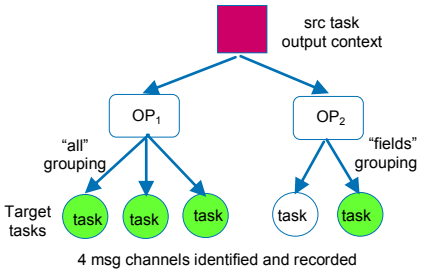

Fig. 8: Reason about output channels using TOC for the example shown in Fig 7.

For re-sending a tuple upon request (through a separate messaging channel) the task selects the buffered tuple with the tuple's *mid* matching the requested *mid*, or the tuple's *mid-set* containing the requested *mid*; but resend the tuple with the single, logically matched *mid*.

Tracking Input Channel in Recipient task. When an input tuple is received, its *mid* or *mid-set* is extracted and an individual *mid* (possibly virtual) that logically matches the recipient task is singled out, that single *mid* is recorded in the inChannelBook, and used in ACK and ASK messages.

During failure-recovery, the restored task, *T*, would ask each source task to resend the possible next tuple wrt the latest one recorded in *T*'s inputChannelBook, thus need to compose a *mid* for the requested tuple guided by its *TIC* and inChannelBook.

5. Experiments

We have built the ***Fontainebleau*** prototype and provided the failure recovery capability based on the architecture and algorithms explained in the previous sections. In this section we briefly overview our experimental results. Our testing environment include 4 Linux servers with gcc version 4.1.2 20080704 (Red Hat 4.1.2-50), 32G RAM, 400G disk and 8 Quad-Core AMD Opteron Processor 2354 (2200.082 MHz, 512 KB cache). One server holds the coordinator daemon, other servers hold workers daemons, each worker supervises several worker processes, and each worker process handles one or more tasks. Based on the topology and the parallelism hint for each logical operation, one or more task instances of that operation will be instantiated by the framework.

5.1. *Correctness of Channel Reasoning*

We first use the experiment results to verify the correctness of channel tracking based on the streaming process topology shown in Fig 1. For simplicity we only have the spout output 100 tuples, which are delivered to "*tr*" tasks in all-grouping, then to "*gemm*" and "*blas*" tasks in fields-grouping, and to "*ana*" and "*agg*" tasks in all-grouping and direct-grouping respectively. Below are some logged information showing the input mid-set, and the resolved mid at the corresponding tasks.

```
-- Task: tr.8
   Received mid: {matrix_spout.10^tr.8-5,matrix_spout.10^tr.9-5}
   Matched mid: matrix_spout.10^tr.8-5

-- Task: gemm.7
   Received mid: {tr.8^blas.0@-32,tr.8^gemm.1@-38}
   Matched mid: tr.8^gemm.1@-38

-- Task: blas.4
   Received mid: {tr.8^blas.0@-12,tr.8^gemm.0@-11}
   Matched mid: tr.8^blas.0@-12

-- Task: ana.11
   Received mid: {gemm.6^ana.11-85,gemm.6^ana.12-85}
   Matched mid: gemm.6^ana.11-85

-- Task: agg.2
   Received mid: blas.4^agg.2-40
                 Matched mid: blas.4^agg.2-40
```

After processing 100 initial input tuples produced by the matrix-spout, the final states of the tasks, including the number of checkpointed tuples (the last ckSeq) as well as the content of InChannelBook and the OutChannelBook, are listed below, and the number of tuples processed by each task is illustrated in Fig. 9. These numbers and states are consistent with the defined semantics of steam processing with the specified grouping criteria.

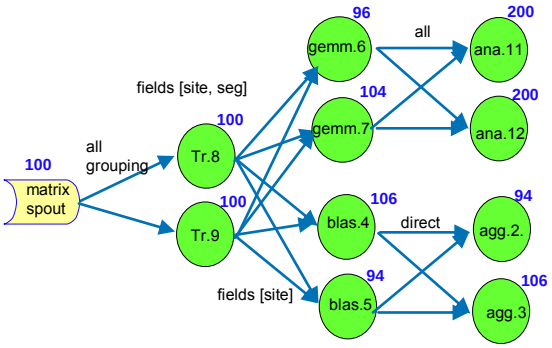

Fig. 9: With 100 input events, the number of tuples processed at each task

For example, with all-grouping, tasks *tr.8* and *tr.9* each gets 100 input tuples from *matrix-spout*, then the 100 tuples output from *tr.8* and the 100 tuples output from *tr.9* are distributed to tasks *gemm.6* and *gemm.7* with each received 96 and 104 tuples respectively, making the total of 200 tuples. Then with all-grouping, each of the derived tuple is further delivered to both tasks *ana.11* and *ana.12*; therefore tasks *ana.11* and *ana.12* each received 200 tuples.

```
++ FINAL tr.8:ckSeq = 100
++ FINAL tr.8:InChannelBook = {matrix_spout.10^tr.8=100}
++ FINAL tr.8:OutChannelBook = {tr.8^blas.1@=47,
tr.8^blas.0@=53, tr.8^gemm.1@=52, tr.8^gemm.0@=48}

++ FINAL tr.9:ckSeq = 100
++ FINAL tr.9:InChannelBook = {matrix_spout.10^tr.9=100}
++ FINAL tr.9:OutChannelBook = {tr.9^blas.1@=47,
tr.9^blas.0@=53, tr.9^gemm.0@=48, tr.9^gemm.1@=52}

++ FINAL blas.4:ckSeq = 106
++ FINAL blas.4:InChannelBook = {tr.9^blas.0@=53,
tr.8^blas.0@=53}
++ FINAL blas.4:OutChannelBook = {blas.4^agg.3=53,
blas.4^agg.2=53}

++ FINAL blas.5:ckSeq = 94
++ FINAL blas.5:InChannelBook = {tr.9^blas.1@=47,
tr.8^blas.1@=47}
++ FINAL blas.5:OutChannelBook = {blas.5^agg.3=53,
blas.5^agg.2=41}

++ FINAL gemm.6:ckSeq = 96
++ FINAL gemm.6:InChannelBook = {tr.9^gemm.0@=48,
tr.8^gemm.0@=48}
++ FINAL gemm.6:OutChannelBook = {gemm.6^ana.11=96,
gemm.6^ana.12=96}

++ FINAL gemm.7:ckSeq = 104
++ FINAL gemm.7:InChannelBook = {tr.8^gemm.1@=52,
tr.9^gemm.1@=52}
```

```
++ FINAL gemm.7:OutChannelBook = {gemm.7^ana.12=104,
gemm.7^ana.11=104}

++ FINAL ana.11:ckSeq = 200
++ FINAL ana.11:InChannelBook = {gemm.7^ana.11=104,
gemm.6^ana.11=96}
++ FINAL ana.11:OutChannelBook = {}

++ FINAL ana.12:ckSeq = 200
++ FINAL ana.12:InChannelBook = {gemm.7^ana.12=104,
gemm.6^ana.12=96}
++ FINAL ana.12:OutChannelBook = {}

++ FINAL agg.2:ckSeq = 94
++ FINAL agg.2:InChannelBook = {blas.5^agg.2=41,
blas.4^agg.2=53}
++ FINAL agg.2:OutChannelBook = {}

++ FINAL agg.3:ckSeq = 106
++ FINAL agg.3:InChannelBook = {blas.5^agg.3=53,
blas.4^agg.3=53}
++ FINAL agg.3:OutChannelBook = {}
```

5.2. *Latency Overhead of Checkpointing*

In the streaming process example shown in Fig 1, the heaviest computation is conducted by tasks of operations "`gemm`" and "`blas`" which are similar so let us focus on "`gemm`". It is the abbreviation for "General Matrix Multiply (GEMM)", a subroutine in the Basic Linear Algebra Subprograms (BLAS) that calculates the new value of matrix *C* based on the matrix-product of matrices *A* and *B*, and the old value of matrix *C,* as

$$C = alpha*AB + beta*C$$

where alpha and beta values are scalar coefficients. GEMM is often tuned by High Performance Computing (HPC) vendors to run as fast as possible, because it is the building block for so many other routines. It is also the most important routine in the LINPACK benchmark. For this reason, implementations of fast BLAS library typically focus on GEMM performance first .

The purpose of this experiment is to exam the impact of checkpoint to the performance of the streaming process involving GEMM operations. For this reason we focus on the performance ratio with and without checkpointing, of a single task. Since multiple tasks may have overlapping disk-writing in checkpointing, measure their overall performance would not give us a clear picture of the above ratio.

We particularly interested in the turning point on the size of input matrices where checkpointing has significant impact to the performance before it, and insignificant impact after it. As in the tuple by tuple stream processing the overall latency is nearly proportional to the number of input tuples, and we measure the performance ratio with and without checkpointing, the impact of the number of input tuples, say from 1K to 1M, is not significant.

In our testing, each original input tuple has 3 two-dimensional N × N matrices of float values, and we measure the above ratio wrt N. Our results shown in Fig 10 indicate that when the matrix dimension size N is smaller than 600, checkpointing has visible impact to the latency of the streaming process; after the matrix dimension size N overpasses 600, that impact becomes insignificant, since in that case the latency is dominated by the computation complexity.

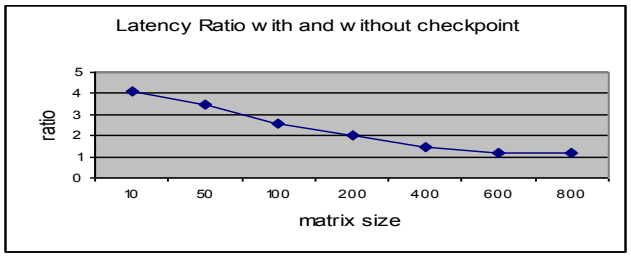

Fig.10: Latency ration with and without checkpoint

5.3. *ASK vs. ACK based Recovery*

Comparing the performance of the ASK-based transactional stream processing with the ACK based one, is the essential motivation of this experiment. In our testing, the failure rate is set to 1%. The matrix dimension size is fixed to 20. With the ACK based approach, a task does not move on to process the next tuple until the result of processing the current tuple has been received, processed and acknowledged by all target tasks; otherwise the tuple will be re-sent after timeout. Therefore the latency overhead is incurred during processing each tuple. Under the proposed ASK based approach, a task does not wait for the acknowledgement to move forward since the acknowledgement is handled asynchronously to the task execution. In this case the latency overhead is only incurred during failure recovery which is rare. Therefore the ASK based approach can significantly improve the overall performance. Our comparison result shown in Fig. 11 has verified this observation.

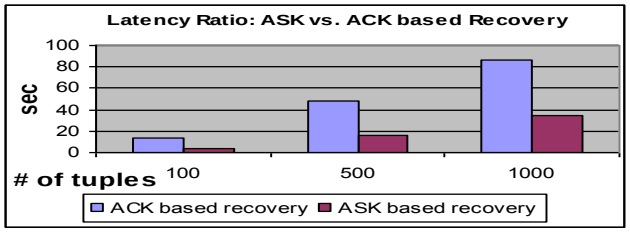

Fig.11: Performance comparison between ACK based and ASK based recovery mechanisms

6. Conclusions

In this work we have taken an initial step to advance the state of art of transactional stream processing with the task-oriented, fine-grinned and backtrack-based failure recovery mechanisms. To provide these mechanisms on top of an existing stream processing platform where message routing is handled by separate system components inaccessible to individual tasks, we tackled the specific hard problem for enabling tasks to track physical messaging channels logically in order to realize re-messaging during failure recovery, for that we introduced the notions of *virtual channel*, *task alias* and *messageId-set*. We also provided a *designated messaging channel*, separated from the regular dataflow channel, for signaling ACK/ASK messages and for resending tuples, in order to avoid interrupting the regular order of data transfer.

These mechanisms have been implemented on our stream analytics platform, *Fontainebleau*, which is built on top of the open-sourced Storm system. With our open station architecture, we ensure all the transactional properties to be system supported and transparent to users. Our experience shows that the novel virtual channel mechanism allows us to handle failure recovery correctly in elastic streaming processes, and the ASK-based recovery mechanism significantly outperforms the ACK-based one.

The proposed solution is integrated to our Live BI platform, a component of our *Igniting Information Insight* strategy with a number of target businesses, which supports the reliable delivery of quality insights and predictive analytics over Big, Fast, Total (BFT) data.

7. References

1. Arasu, A., Babu, S., Widom, J. The CQL Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal, (15)2, June 2006.
2. D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In CIDR, 2005.
3. Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker, “FaultTolerance in the Borealis Distributed Stream Processing System”, SIGMOD 2005.
4. Irina Botan, Peter M. Fischer, Donald Kossmann, Nesime Tatbu, “Transactional Stream Processing”, EDBT 2012.
5. David B. Johnsonandwillyzwaenepoel, “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing”, Journal of Algorithms 11, 462-491 (1990).
6. Qiming Chen, Meichun Hsu, Hans Zeller, “Experience in Continuous analytics a s a Service”, EDBT 2011.
7. Qiming Chen, Meichun Hsu, “Experience in Extending Query Engine for Continuous Analytics”, Proc. 11th Int. Conf. DaWaK, 2010.
8. Qiming Chen, Meichun Hsu, “Query Engine Net for Streaming Analytics”, Proc. 19th International Conference on Cooperative Information Systems (CoopIS), 2011.
9. D.J. DeWitt, E. Paulson, E. Robinson, J. Naughton, J. Royalty, S. Shankar, A. Krioukov, “Clustera: An Integrated Computation And Data Management System”, *VLDB* 2008.
10. Michael J. Franklin, et al, “Continuous Analytics: Rethinking Query Processing in a NetworkEffect World”, CIDR 2009.
11. Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, Myung Cheol Doo, “SPADE: The System S Declarative Stream Processing Engine”, *ACM SIGMOD 2008*.
12. J.-H. Hwang, M. Balazinska, et al. High-Availability Algorithms for Distributed Stream Processing. In Proc. of ICDE'05, Washington, DC, USA, 2005.
13. D. B. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *J. of* Algorithms, vol. 11, pp. 462-491, 1990.
14. J. Li and A. Karp. Access control for the services oriented architecture. In ACM Workshop on Secure Web Services, 2007.
15. M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly Available, Fault-tolerant, Parallel Datafows. In Proc. of SIGMOD, New York, NY, USA, 2004.
16. Tweeter, “Transactional topologies”, https://github.com/nathanmarz/storm/wiki/Transactional-topologies, 2012.