Backtrack-Based and Window-Oriented Optimistic Failure Recovery in Distributed Stream Processing

Qiming Chen(✉), Meichun Hsu, and Malu Castellanos

HP Labs, Palo Alto, CA, USA
{qiming.chen,meichun.hsu,Castellanos.malu}@hp.com

Abstract. Support transaction property and fault-tolerance is the key to applying stream processing to industry-scale applications; however the corresponding latency overhead must be minimized for accommodating real-time analytics. This issue has been studied in various contexts. In this work we develop the ***backtrack failure recovery*** mechanism to allow a task to roll forward without waiting for acknowledgement from its downstream target tasks in the failure-free case, but to request its upstream source tasks to resend the missing tuples only during failure recovery which is the rare case thus has limited impact on the overall performance. For further reduced latency we extend our solution in another dimension by applying the notion of ***optimistic checkpointing*** to stream processing, and propose the **Continued** stream processing with ***Window-based Checkpoint and Recovery*** (CWCR) approach, allowing a task to emit results tuple by tuple continuously but checkpoint in batch, and acknowledge, only once per window (e.g. time window). We also tackle the hard problems found in implementing a transactional layer on-top of an existing stream processing platform. We have implemented the proposed mechanisms on *Fontainebleau*, the distributed stream analytics infrastructure we built on top of the open-sourced Storm platform. Our experiment results reveal the novelty of the proposed technologies and the feasibility to support fault-tolerance with minimal latency overhead for real-time stream processing.

Keywords: Stream processing · Failure recovery · Dataflow transaction · Pessimistic checkpointing · Optimistic checkpointing

1 Introduction

To apply stream processing to industry-scale applications, ensuring transaction property and fault-tolerance is the key issue. A stream processing application is modeled as a continued dataflow process - ***streaming process***, where the parallel and distributed tasks are chained in a graph-structure with each task transforming a stream to a new stream. The transaction property guarantees the streaming data, called *tuples*, to be processed in the order of their generation in every dataflow path, with each tuple processed once and only once, under the notion of *eventual consistency* [3, 15, 17, 18].

M. Castellanos et al. (Eds.): BIRTE 2013 and 2014, LNBIP 206, pp. 38–64, 2015.
DOI: 10.1007/978-3-662-46839-5_4

1.1 Prior Art

There exist multiple lineages of transactional stream processing. The first thread of work is characterized by applying the database transaction semantics to unbounded data stream with the concept of *snapshot isolation* [2, 4, 6, 7, 10], namely, to split a stream into a sequence of bounded chunks and handle each chunk in a transaction boundary. In this way, processing a sequence of data chunks generates a sequence of state snapshots. However, this mechanism only cares about the *state oriented* transaction boundary without addressing failure recovery.

The second thread is originated from reliable dataflow based on message logging and resending for failure recovery. One current approach, represented by Storm's "transactional topology" [18], treats the whole stream processing topology as a single operation thus suffers from the loss of intermediate results in the occurrence of failures. Another limitation of this approach is the ignorance of the states of data buffered in tasks.

The third thread is based on checkpointing and *forward tracking* where a task, T, checkpoints each output tuple, t, before emitting it, then T waits for the target tasks (recipients) to confirm (by acknowledgement - ACK), the success of processing t, before emitting the next output tuple; if T does not receive the ACK after a timeout (e.g. in case the target task fails, it takes a while to be restored), T will resend t, again and again, until being acknowledged [14, 15, 17]. Although the "once and only once" semantics can be enforced by ignoring duplicate tuples, waiting for ACK and keeping resending on the per tuple basis cause extremely high latency.

The forth thread is the variation of the above approach; it is also based on checkpointing and message resending, but characterized by "***backward tracking***", namely, allowing a task to process tuples continuously without waiting for acknowledgements and without resending tuples in the failure-free case, but to request (with the **ASK** message) 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. Our experience shows that compared with *forward tracking*, the *backward tracking* approach can reduce the overall latency of stream processing significantly. However, it still suffers from the overhead of per-tuple checkpointing.

The fifth thread, referred to as ***optimistic checkpointing***, also focuses on reducing the latency in failure free case [13, 19]. Unlike per-tuple based *pessimistic checkpointing* [16, 19], optimistic checkpointing allows messages to be checkpointed occasionally in batch. However, in general distributed computing, the use of this mechanism may cause uncontrolled task rollbacks known as the domino effects [19].

1.2 Proposed Approach

In the context of graph-structured, distributed stream processing, the checkpoint based failure recovery approaches discussed so far are in general limited to pessimistic and forward tracking ones. In this work we take the initial step to ***combine optimistic checkpointing*** and ***backward tracking failure recovery***, which allows us to gain the benefits of both for low-latency, real-time stream processing.

With the ***backtrack failure recovery*** mechanism, a task does not wait for ACK and re-emit output on the per-tuple basis, but requests the missing tuples from its upstream

source tasks only in failure recovery which is the rare case thus has limited impact on the overall performance.

For further reduced latency we extend our solution in another dimension – instead of per-tuple based *pessimistic* checkpoint protocol, we adopt the ***optimistic checkpoint protocol*** under the criterion of *eventual consistency* [3, 15], which allows the checkpoints to be made occasionally in batch and asynchronously with tuple processing and emitting. We solve the uncontrollable rollback problem found in the context of instant consistency of global state, by associating checkpoint boundary with the window semantics of stream processing to provide the commonly observable and semantically meaningful synchronization point of task rollbacks. We propose the ***Continued stream processing with Window-based Checkpoint and Recovery*** (**CWCR**) approach, under which the stream processing results are emitted tuple by tuple continuously (thus different from batch processing), but checkpointed once per-window (typically time-window). Since the failure recovery is confined in the commonly observable window boundaries, the so called domino effect in chained rollbacks can be avoided even in the situation of failure in failure.

To implement the proposed transaction layer on-top of an existing stream processing platform we need to deal with the hard problem on how to keep track the input/output messaging channels in order to realize re-messaging during failure recovery. Since common to the modern component-based distributed infrastructures, the data routing between tasks is handled by separate system components inaccessible to individual tasks, making it trivial for tasks to track. 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 hub***, 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 flow.

We have implemented the proposed mechanisms on *Fontainebleau*, the distributed stream analytics infrastructure we develop on top of the open-sourced Storm platform. Our experiments reveal the novelty and value of these mechanisms. The combination of optimistic checkpointing and backtrack recovery significantly reduces the latency of transactional stream processing thus making it feasible for real-time applications; and the virtual channel mechanism allows us to handle failure recovery correctly in the elastic stream processing infrastructure.

The rest of this paper is organized as follows: Sect. 2 outlines the concept of graph-structured distributed streaming process and out platform; Sect. 3 describes backtrack failure recovery; Sect. 4 discusses Continued stream processing with Window-based Checkpoint and Recovery (CWCR); Sect. 5 discusses how to track messaging channels intelligently; Sect. 6 illustrates the experiment results; Sect. 7 concludes.

2 Distributed Stream Processing Infrastructure

2.1 Graph Structured Streaming Process

A stream is an unbounded sequence of events, or tuples. Logically a stream processing operation is a continuous operation to apply to the input stream tuple by tuple to derive

a new output stream. In a distributed stream processing infrastructure, 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; in each cycle it processes an input tuple, updates the execution state and emits the resulting tuples. Tuples transmitted between tasks are carried in messages,

Let us observe a streaming process example for matrix manipulation based event analysis. In this streaming process, the source tuples are streamed out, with second-based timestamps, 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 the tasks of operation "*tran*" for transformation, then to "*gemm*" (general matrix multiplication) and "*trmm*" (transpose and multiplication) with "fields-grouping" on different hash keys; the outputs of "*gemm*" tasks are distributed to "*ana*" (analysis) tasks with "all-groupuing", and the outputs of "*trmm*" tasks are distributed to "*agg*" (aggregation) tasks with "fields-grouping". The logical operations, links and grouping types are illustrated in Fig. 1 and specified below.

```
BlueTopologyBuilder builder = new BlueTopologyBuilder();
builder.setSpout("matrix_spout", matrix_spout, 1);
builder.setBolt("tran", tran, 4).shuffleGrouping("matrix_spout");
builder.setBolt("gemm", gemm, 2).fieldsGrouping("tran", new Fields("site", "seg"));
builder.setBolt("ana", ana, 2).allGrouping("gemm");
builder.setBolt("trmm", trmm, 2).fieldsGrouping("tran", new Fields("site"));
builder.setBolt("agg", agg, 2).fieldsGrouping("trmm", new Fields("site"));
```

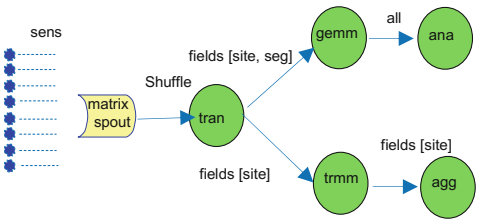

Fig. 1. A logical streaming process

Physically, each operation has more than one task instances. Given a pair of source and target operations, say *trans* and *gemm*, the tuples transmitted between their tasks are grouped with the same criteria defined on the operation level, as illustrated in Fig. 2. The possible input and output channels of a task can be extracted from the streaming process topology statically, but the actual channels used in distributing an emitted tuple are resolved dynamically during execution according to the grouping type, tuple content (e.g. fields hash value) and loading balance.

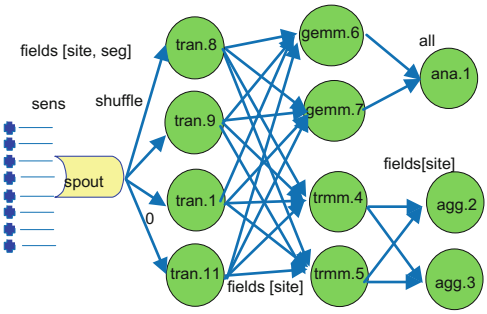

Fig. 2. A physical streaming process where an operation is instantiated as multiple tasks

2.2 Distributed Stream Processing Platform

We have developed a *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 – an open sourced 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, that provides canonical system support but open for plugging in application logic. In the OO programming context, an open-station is provided with the open-executor coded by invoking certain abstract methods to be implemented by users based on their application logic.

We treat transaction enforcement with failure recoverability as a kind of task execution pattern, and provide the corresponding open stations hierarchy for supporting it automatically and systematically. The basic transactional station is defined as an abstract class *BasicCkStation* with several major system methods:

- *prepare* – invoked when the task is initially setup or restored from a failure, therefore the *recovery*() method discussed below is invoked, when necessary, inside the prepare() method.
- *execute* – invoked cycle by cycle for per-tuple processing, checkpointing, acknowledging, emitting, etc.
- *recovery* – invoked for recovery, including rolling back to the last checkpoint, re-emit output to the right target tasks at the downstream, and re-acquire the latest input from the source task at the upstream, etc.
- *outputFields* – used to specify the fields of the output tuples.

These methods invoke certain abstract methods which will be implemented based on application specific semantics (e.g. setup initial state, processing a tuple) and resource specific properties (e.g. the specific checkpoint engine based on files or databases). 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).

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.

2.3 Backtrack Based Failure Recovery

The checkpointing based failure recovery in stream processing is typically characterized by ***forward tracking*** or **ACK**-based, i.e. a task cannot emit the next tuple until the successful processing of the last emitted tuple is acknowledged; and if such ACK is not received in timeout the task must resend the last tuple again and again, until being acknowledged. Although the "once and only once" semantics can be enforced by ignoring duplicate tuples, waiting for ACK and re-emitting output on the per tuple basis causes extremely high latency.

For enhanced overall performance, we stick on the ***backtrack*** or **ASK**-based recovery approach, where a task does not wait for acknowledgement before moving forward; instead, acknowledging is asynchronous to task executing and only used to remove the buffered tuples already processed by the target tasks.

Specifically, a task keeps an emitted tuple in a pool until there is no need to resend it for failure recovery; the task can determine the fan-out of an emitted tuple based on the topology, and detect whether that tuple is fully processed (thus fully acknowledged) by all the target tasks it was distributed to. A fully acknowledged tuple can be removed from the message pool. Since acknowledgement is only used to trigger the removal of the acknowledged tuple and all the tuple prior to that tuple, any ACK is allowed to be lost.

During failure recovery, the reestablished task tells each source task the last tuple it has seen, and **ask**s the source task to resend the next one; and if not received after timeout it would ask again and again. Since failures are rare, such backtrack process has limited impact on the overall performance.

2.4 Separated Message Hub for Recovery

A hard problem in supporting message resending on top of an existing stream processing platform is how to ensure the order of regular tuple delivery not interrupted by the task recovery process, when there lacks the accessible message re-sorting facility [14, 15, 17]. 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 hub, separated from the regular dataflow channel, is provided for a task, for signaling ACK/ASK and resending tuples (Fig. 3).
- 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 hub 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 hub. 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.

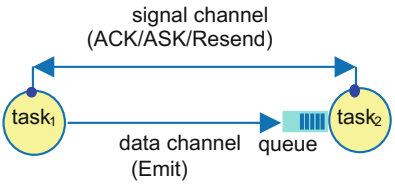

Fig. 3. Secondary messaging hub for ACK/ASK and resend

Checkpoint based failure recovery requires a task to know exactly the messaging channel and record it before emitting a tuple. For now we assume this requirement is satisfied. We will present our channel resolution mechanism later in Sect. 5.

3 Pessimistic Checkpointing with Backtrack Recovery

In stream processing, the typical checkpointing protocol is ***pessimistic*** where every output message (carrying resulting tuple) of a task is checkpointed before sent. Recovery based on pessimistic checkpointing is relatively simple since for each input channel only one possible missing tuple is concerned.

As mentioned above, for enhanced overall performance, we stick on the *backtracking* based failure recovery, where a task continuously emits resulting tuples without waiting for ACKs, but requests the missing tuples only after reestablished from a failure (by ASK message) from its source tasks. In this section we discuss pessimistic checkpointing incorporated with the *backtracking* based failure recovery.

Task Execution. A task runs cycle by cycle continuously for processing input 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 ("jumped") 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. After the tuple is processed, the resulting state, the input message-id, the output messages (holding tuples), etc., are checkpointed (serialized and persisted to file); the transaction is "committed", acknowledged and the output messages are emitted. The algorithm of ***execution()*** is outlined in Fig. 4.

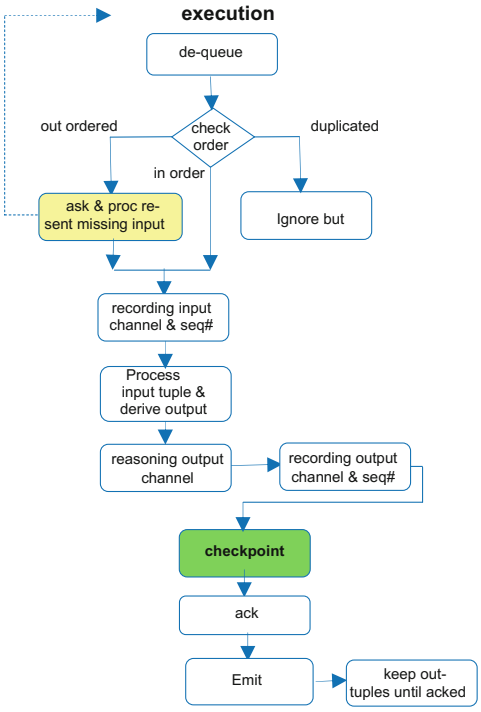

Fig. 4. Task execution with pessimistic checkpointing

Task Recovery. Supported by the underlying Storm platform, a failed task instance is 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, again and again until received
- re-emit the last output tuple; in case that tuple has not been lost, as a duplicate tuple it will be ignored by recipient tasks.

When the recovering 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 and recorded in its input-map that is a part of

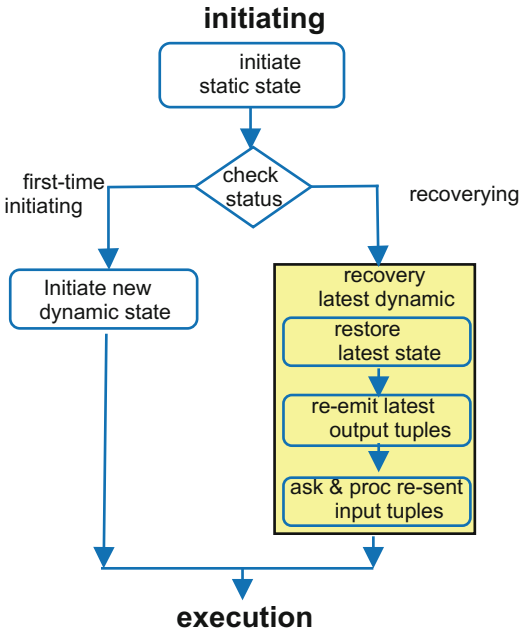

Fig. 5. Task initiate/recovery with pessimistic checkpointing

the checkpoint content. The resent tuples are processed first before the restored task entering the regular data processing loop. If a resent tuple has already transmitted to the input queue, then the queued tuple will be identified as duplicated and ignored.

Due to the disk access overhead of per-tuple checkpointing and the messaging overhead in acknowledging each tuple, the pessimistic protocol is inefficient in the failure-free case; even if incorporated with the backtrack recovery. As a result, it cannot satisfy the latency requirement for real-time stream processing. To make transactional stream processing a feasible and acceptable technique we need to explore another kind of checkpointing protocol – the *optimistic* checkpointing protocol.

4 Optimistic Checkpointing with Backtrack Recovery

4.1 Concept and Problem

In the modern computing environment, failures are infrequent and the overall performance is primarily contributed by the failure-free performance, which has motivated us to investigate the failure recovery mechanism based on ***optimistic*** checkpointing.

Unlike the pessimistic checkpointing mechanism with which the checkpointing is synchronized with the per-tuple processing, the optimistic checkpointing mechanism is characterized by checkpointing *occasionally* and *asynchronously* with the per-tuple processing. Under this mechanism, a task checkpoints once after processing multiple tuples, while still emits result tuple by tuple continuously. As a result, the failure-free performance

can be significantly improved due to reduced disk access, and the real-time feature can be retained since the per-tuple processing is not blocked by checkpointing.

However, when a task is restored from a failure with its state being rolled back to the last checkpoint, the effects of processing multiple tuples would be lost and should be redone; for that the task would request its source tasks to resend multiple tuples emitted since then. These resent tuples are processed in advance, before the restored task goes to the regular per-tuple processing loop, towards the "*eventual consistency*" [3, 11].

It is worth noting the difference between optimistic checkpointing and batch processing. With optimistic checkpointing, although the output tuples of a task is checkpointed in batch, they are emitted tuple by tuple continuously in real-time stream processing.

The notion of optimistic checkpointing was previously studied in the context of general distributed systems where the instant consistency of globally state is of the primary concern [16, 19]. In that case rolling back a task may cause any other task to rollback until a consistent global state has been reached; if without a commonly observable and semantically meaningful synchronization point for "cutting off" rollback propagation, it may eventually lead to the domino effect – an uncontrolled propagation of task rollbacks.

The concept of synchronization point for rollback propagation can be explained by the following example. Assume a pair of source and target tasks T_A and T_B checkpoint their states/messages per 100 input tuples respectively, and out of one input tuple T_A derives 4 output tuples and sends them to T_B. As mentioned before a checkpoint state includes the information about input/output messages and computation results. Considering the following situation after T_A and T_B start running simultaneously:

(a) After T_B processed 100 input tuples received from T_A since T_B's last checkpoint, it persists its state into a new checkpoint p_b. By then, however, T_A only processed 25 input tuples since its last checkpoint, with the result not being checkpointed. In this case, we say that the checkpoint p_b is an ***unstable checkpoint*** since it is not supported by the checkpoint history of T_A.
(b) Assume that T_B failed after processing some tuples since point (a), it is restored and rolled back to p_b and tends to request T_A to re-send the missing tuples. In case T_A also failed and lost the un-checkpointed history of output tuples since its last checkpoint, T_A cannot identify the tuples requested by T_B.
(c) As a result, both T_A and T_B must further rollback to a possible common synchronized point. Such rollback propagation is uncontrolled; in the worst case, both tasks have to roll back to the very beginning.

It can be seen from the above example that in order to apply optimistic checkpointing to the failure recovery of stream processing, in addition to adopting the notion of "*eventual consistency*" [3, 11], it is necessary to provide a commonly observable and semantically meaningful synchronization point to avoid uncontrolled rollback propagation. We solve this problem by incorporating the concept of synchronization point with the window semantics of stream processing.

4.2 Window Semantics of Stream Processing

Although a data stream is unbounded, very often applications require those infinite data to be analyzed granularly. Particularly, when the stream operation involves the aggregation of multiple events, for semantic reason the input data must be punctuated into bounded chunks. This has motivated us to execute such operation *epoch by epoch* to process the stream data *chunk by chunk*, and this has given us the fitted framework for identify the synchronization points for optimistic checkpointing.

For example, if the stream contains time-series data with timestamps (e.g. per second), and an operation aims to deliver time-window (e.g. per minute) based aggregation, then the execution of this operation on an infinite stream is made in a sequence of *epochs*, one on each stream chunks falling in that time-window. In general, given an operator, O, over an infinite stream of relation tuples S with a criterion ϑ for cutting S into an unbounded sequence of chunks,

$$< S_0, S_1, \ldots, S_i \ldots >$$

where S_i denotes the *i-th* "chunk" of the stream according to the chunking-criterion ϑ. The semantics of applying O to the unbounded stream S lies in

$$O\,(S) \rightarrow < O\,(S_0), \ldots O(S_i), \ldots >$$

which continuously generates an unbounded sequence of results, one on each *chunk* of the stream data.

The *paces* of dataflow wrt timestamps can be different at different operators; for instance, an operation for hourly aggregation has a larger pace then the operation for per-minute aggregation.

Punctuating input stream into chunks based on a window-boundary is a template behavior common to many stream operations, thus we consider it as a kind of meta-property of stream operations and support it systematically. To handle window boundary in a task, a base unit, τ (e.g. 1 min or 1 tuple) is required, and the following three variables are defined:

- w_{delta}: the window size by τ, e.g. 1 min, or 100 tuples;
- $w_{current}$: the current window sequence number; e.g. 1 for the window of minute 1, 2 for the window of minute 2, … etc.;
- $w_{ceiling}$: the starting sequence number of the next window by number of τ; e.g. 2 for the window of minute 2; 201 for the window of 200–300 tuples.

Then several functions are defined on each input tuple $\boldsymbol{t}$ to determine whether $\boldsymbol{t}$ is within or beyond the current window, e.g.

- $fw_{current}\ (t)$: returns the $w_{current}$; e.g. if t contains timestamp 100 (by second), the current window number (by minute) for t is 2.
- $fw_{next}(t)$: returns a boolean for detecting whether t belongs to the next window, and updates $w_{current}$ and $w_{ceiling}$ as appropriate.

4.3 Window-Based Checkpoint and Recovery

Aiming to support failure recovery by optimistic checkpointing, we incorporate the notion of synchronization point with the window semantics of stream processing, and propose the protocol of ***Continued stream processing with Window-based Checkpoint*** (**CWCR**), which allows a task to process data and emit results continuously on the per-tuple basis, but checkpoints state on the per-window basis.

With CWCR, the checkpointing is made once per-window (typically time-window) *asynchronously* with per-tuple execution. When a task T is re-established from a failure in a window boundary w, its last checkpointed state is restored; and the tuples T received from the beginning of w up to the most recent ones, from all input channels, are requested and resent to T. These resent tuples are processed in the recovery phase before T goes to the regular stream processing loop.

Compared with the pessimistic checkpointing approach, the benefit gained from CWCR consists in the enhanced overall performance by avoiding the overhead of per-tuple checkpointing in the absence of failures.

4.4 CWCR Synchronization Point

To describe CWCR more formally, let us denote a checkpoint of task T by S_T; denote the input *messageIds* (*mids*) and the output messages contained in S_T by μS_T and σS_T respectively; and denote the checkpoint history of T by ηS_T. Further, given a pair of source and target tasks A and B, the *messages* from A to B in σS_A and ηS_A are denoted by $\sigma S_{A\to B}$ and $\eta S_{A\to B}$ respectively; the input *mids* to B from A in μS_B and ηS_B are denoted by $\mu S_{B\leftarrow A}$ and $\eta S_{B\leftarrow A}$ respectively. Based on these notations we define the following concepts.

- **Checkpoint History:** the sequence of checkpoints of task T is referred to as T's checkpoint history.
- **Stable Checkpoint:** a checkpoint is *stable* if it can be reproduced from the checkpoint history of its upstream neighbors. More precisely, the checkpoint of task B. S_B, is *stable* wrt a source task A iff all the messages identified by $\mu S_{B\leftarrow A}$ are contained (denoted by $\propto$) in $\eta S_{A\to B}$, i.e.

$$\mu S_{B\to A} \propto \eta S_{A\to B}.$$

  S_B is *stable* iff S_B is *stable* wrt all its source tasks.
- **Backtrack Recoverability:** given a pair of source and target tasks A and B, task B is backtrack recoverable from a failure wrt task A, if since B's last checkpoint, all the input tuples from A can be resent by A, even if in the case that A also fails.

Then we compare the task recoverability wrt pessimistic and optimistic checkpointing.

Recoverability Rule with Pessimistic Checkpointing
With pessimistic checkpointing a task is backtrack recoverable if every input tuple is checkpointed by the corresponding source task ***before emitting***.

A task with pessimistic checkpointing follows the above rule by first checkpointing and then emitting each output tuple, which ensures that the missing tuple during a

failure can always be found and resent, even if the source task also fails. This is the strong criterion for recoverability.

With optimistic checkpointing, the above strong recoverability rule cannot be followed because the output tuples of a task are emitted continuously but checkpointed only occasionally; therefore we need to find a relaxed recoverability rule.

Intuitively, if task is reestablished from a failure and rolled back to its last checkpoint say p, to guarantee that the each missing tuple since p can be figured out and resent by the corresponding source task even if that source task also fails, p must be a stable checkpoint. This forms the basis of the following rule.

Recoverability Rule with Optimistic Checkpointing
With optimistic checkpointing a task B is backtrack recoverable iff B's checkpoints are stable, i.e. for each source task A of B, $\mu S_{B \leftarrow A} \propto \eta S_{A \rightarrow B}$.

Ensure checkpoint stability is the key to avoid the domino effects in optimistic checkpoint based task recover. With CWCR we incorporate this with the window based chunking criterion. Specifically, for time series data, we provide a timestamp attribute for the stream tuples, and use a time window, such as per minute time window, as the basic checkpoint interval (although the concept of window is not limited to time window).

For example, given a pair of source and target tasks T_A and T_B, if the checkpoint interval of T_A is w_{delta} and that of T_B is $N \times \boldsymbol{w}_{delta}$ where N is an integer, then the checkpoint of T_B is stable wrt T_A. For instance, if the checkpoint interval of T_A is per minute (60 s), and that of T_B is 1 min (60 s), 10 min (600 s) or 1 h (3600 s), then T_B's checkpoint is stable wrt T_A; otherwise if T_B's checkpoint interval is 90 s, it is not stable wrt to T_A, and in that case if T_B rollback to its latest checkpoint and requests T_A to resend the missing messages, there is no guarantee for T_A to identify and find them.

Based on these concepts we provide the algorithms for CWCR based failure recovery algorithms.

4.5 CWCR Algorithms

Overview. With the CWCR mechanism, a task, T, runs cycle by cycle to process stream data tuple by tuple. In each cycle before reaching a window boundary T takes an input tuple, records the *mid*, processes the tuple, records and emits the output tuples, but without checkpoint and acknowledgement. When a window boundary is reached, the task T checkpoints the current state, the latest seq# of all input channels, and all the output tuples as a *single checkpoint*; then T acknowledges each source task *only once* (per window) with the latest seq# in the corresponding input channel.

During failure recovery, a task T rolls back to the last checkpoint (at the end of the last window), then it sends *one message* to each source task, say T_s, asking for all the tuples T_s emitted to T since then in the current window boundary, then T_s would resend T these tuples in a single message through the *signal messaging hub* that is separate from the dataflow channel as explained above. In case T does not receive the resent tuple it would ask again, and again until receives. Then T will reprocess all the resent tuples first, before going to the regular stream processing cycles. Later the input tuples duplicated with the resent ones will be ignored.

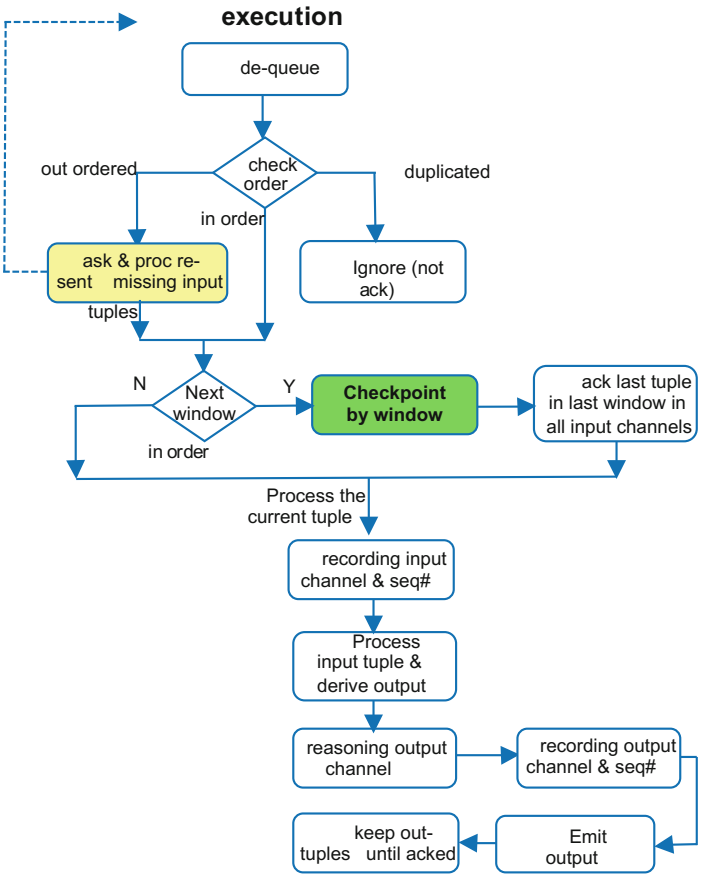

Fig. 6. Task execution with CWCR

A checkpoint contains information about

- the task-id, topology-id, number of tuples processed,
- the checkpoint window boundary - $\boldsymbol{w}_{current}, \boldsymbol{w}_{ceiling}$;
- the latest seq# of all input/output channels (kept in Maps M_i and M_o) from which the latest input/output *mids* in those channels can be derived),
- the current (final) computation state wrt the current window
- the output tuples emitted in the current window.

As mentioned above, a task, as an iteratively executed program for processing stream data tuple by tuple, is provided with two major methods: the ***prepare()*** method for instantiating the system facilities and the initial state, and the ***execute()*** for processing an input tuple in the main stream processing loop. Failure recovery, ***recovery()***, is handled in *prepare()* since after a failed task restored it will experience the *prepare()* phase first. There exist other functions for interpreting the ACK for removing the pooled tuples no longer needed, and ASK for resending tuples.

Task Execution. The algorithm of ***execute()*** is illustrated in Fig. 6 with the following major steps.

- Resolve t's message id (*mid*), channel (c) and *seq#* (k_{curr}) of t, which may be virtual (described later) (*line 1*).
- Compare t's seq# (k_{curre}) and the expected seq# ($k_{last} + 1$); if t is duplicate (with smaller seq# than expected) it will not be processed again, i.e. ignored; 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 (*line 2–8*).
- Update M_i with the current input channel and seq# (*line 9*).
- Given an input tuple t, $fw_{next}(t)$ is applied to check whether t falls into the next window boundary, and if true, the task T makes a single checkpoint for the state of the entire data processing in the current window, then T acknowledges each source task *only once* with the latest seq# in the corresponding input channel (*line 10–18*).
- Processing t and generating output tuples L_{out} (*line 19*).
- For each resulting tuple t_{out} in L_{out}, resolve its mid_{out} that may be virtual, embed mid_{out} in t_{out}, compose the carrying message m_{out}, and append m_{out} to the list of output messages L_{out_msg} (*line 21–26*).
- Emitting the output tuples (*line 27*).
- Pooling the output tuples (*line 28*), aiming to resend upon request. The pooled tuples will be removed upon acknowledgement on the per-channel basis where an ACK with seq# k causes the garbage collection of all the pooled tuples with seq# $\leq k$.
- Advance execution *cycle#* (*line 30*).

Algorithm 1. Task Execution (Tuple t)

```
1  extract mid, channel c, seq# k_curr from t
2  k_last ← get_seq (M_i, c);
3  if k_curr > k_last + 1 then
4     do_miss (c, k_last, k_curr);
5     k_last ← get_seq (M_i, c)
6  else if k_curr ≤ k_last then
7     return;
8  end if
9  put_seq(M_i, c, k_curr);
10 if (fw_next(t)) then
11    ck (S_ck)
12    ∀ c ∈ C_i do
13       mid_ack ← get_mid (c, M_i)
14       if mid_ack ≠ ∅ then
15          ack (mid_ack);
16       end if
17    end do
18 end if
19 L_out ← processing(t);
20 if L_out ≠ ∅ then
21    ∀t_out ∈ L_out do
22       mid_out ← getset_out_mid(M_o, t_out);
23       t_out ← embed (t_out, mid_out)
24       m_out ← compose (t_out, mid_out)
25       L_out_msg ← add (L_out_msg, m_out)
26    end do
27    emit_all(L_out_msg)
28    L_pool_msg ← append (L_pool_msg, L_out_msg)
29 end if
30 cycle++
```

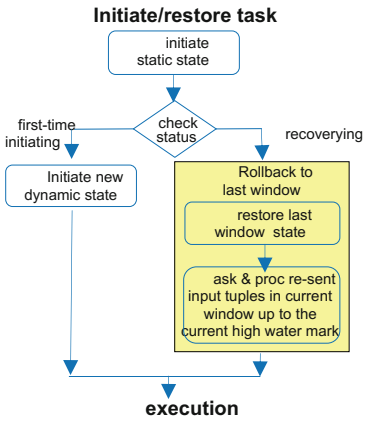

Fig. 7. Task recovery with CWCR handled in prepare()

Task Recovery. Supported by the underlying infrastructure, a failed task instance can be re-initiated on an available machine node by loading the serialized task class to the selected node and new an instance over there.

The algorithm of ***recovery()***, as illustrated in Fig. 7, includes the following steps.

- The state S_{ck} of task T is restored from the checkpoint of last window (*line 1*).
- T sends one RESEND request to each source task in all the possible input channels C_i, asking for resending inputs starting from the first tuple (by *seq#*) of the current window; then T processes the resent tuples channel by channel sequentially before going to the normal execution loop (*line 2–10*).

In each source task, say T_s, upon receipt the above request, gets the latest, not yet emitted output seq#, t_h, and resends to T all the output tuples with seq# up to t_h *in a single message* from the dedicated signal messaging hub.

Algorithm 2. Task Recovery (ck_bytes)

```
1  S_ck ← restore(ck_bytes)
2  ∀ c ∈ C_i do
3      mid_ask ← get_mid (c, M_i)
4      if mid_ask ≠ ∅ then
5          L_resent ← ask_wait (mid_ask)
6          ∀ t ∈ L_resent do
7              exec(t)
8          end do
9      end if
10 end do
```

4.6 Failure in Failure

Recovering backwards chained failures is a recursive process, and we need a boundary condition – the event source S that feeds data stream to a streaming process topology, must be reliable, i.e. when requested, S can re-send the current tuple (wrt pessimistic

checkpoint) or the tuples in the current window (wrt optimistic checkpoint) to the topology.

With pessimistic checkpointing, given a chained source and target tasks A and B, if B fails and restored, B would keep asking A to resent the last tuple; if A also fails, it will recover itself first, and after re-established it can always find and resend the requested tuple by B since any emitted tuple is checkpointed.

With optimistic checkpointing, the re-messaging in recovering single point failure and chained failures are different although the eventual results are consistent. Refer to Fig. 8, let us consider event source S and chained tasks A, B, C; in $window_1$ all the tasks processed 5 tuples, and in $window_2$ A processed input s_6, s_7, s_8 from S and output $a_6, a_7, a_8 \ldots$ etc. Assume task B fails after processing a_6, a_7, a_8 and emitting b_6, b_7, b_8, task B would ask A to resend the tuples in $window_2$ starting from a_6. In case A does not fail, A would resend them.

However, if task A also fails before resending, the following would happen.

- Task A must recover itself first by asking S to resend the tuples in $window_2$, eventually S re-emits s_6, s_7, s_8 to A for processing.
- As a result of A's recovery, A has already re-emitted a_6, a_7, a_8 to B through the regular data channel, which is recorded with A, but kept in B's in-queue without being processed before B receives and processes the resent tuples explicitly requested.
- Then in processing B's resend request, A would identify these already emitted tuples and resend them through the signal/resend channel. B will process the resent tuples and later ignored the previously re-emitted tuples.
- Since before B's failure, B might have emitted b_6, b_7, b_8 to task C, the same tuples emitted after failure recovery would be ignored by C.

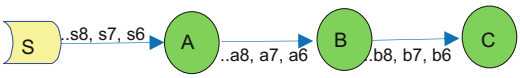

Fig. 8. Recover chained failures

The above system behavior allows us to deal with failure-in-failure correctly. Since this situation is very rare, the overhead in additional messaging is insignificant.

In summary, compared with the pessimistic checkpointing based recovery approach, CWCR allows a task to checkpoint and acknowledge only once per window rather than on the per-tuple basis, but needs to process more resent tuples during failure recovery (although in each channel the resent tuples are carried by a single message), which constitute a beneficial performance trade-off in environments where failures are infrequent and failure-free performance is of primary concern. Further, under CWCR a task still emits output tuple by tuple continuously; therefore the latency requirement for real-time stream processing is retained. Since CWCR relies on window boundaries to synchronize the checkpoints of chained tasks to avoid the so called domino effects, therefore making the rollback propagation well controlled.

5 Tracking Message Channels Intelligently

5.1 The Messaging Channel Tracking Problem

In order to build a transaction layer on top of an existing parallel and distributed stream processing framework like Storm, rather than re-develop a new underlying framework 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*** (***mid***) 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, to record the message-id ***before*** sending a message, in order to resend the right message to the right target task if the previously sent message is missing.

The challenge 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, resulting the following ***messaging channel paradox***.

- 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 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, according to its records, 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.

Our solution to these problems is characterized by ***tracking 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.

As mentioned previously, we ensure the regular order of data delivery not to be interrupted by the failure recovery process by providing a ***designated messaging hub*** that is separated from the regular dataflow channel, for signaling ACK/ASK and resending tuples.

5.2 Basic Notations

We first define the following notations.

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

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

5.3 MessageId-Set and Virtual MessageId

We consider two kinds of "logical message identifiers", one for a set of recipients, another for 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, from the *mid-set* the target task picks up the *mid* with target taskId matching its own taskId, and records the corresponding input channel and seq#. This matched *mid* will be used for identifying both ACK and ASK messages. In the other words, *mid-sets* are recorded in the source task only with the output tuples to be sent; in the target task, only the matched single *mid* is recorded and used. A task identifies the buffered tuple matching the *mid* carried by an ACK or ASK message 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; the sending task has no knowledge about the physical destination before emitting a tuple but can calculate that number, and can treat that number as the ***alias*** of the corresponding target task ID, and use the *alias* as the target task to create a ***virtual mid***. A virtual mid is directly recorded and used in both the sending and receiving tasks.

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 an ACK or ASK message identified by a single *mid*, and a kept 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 is similar to have the Map results sent to the 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, and 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 the example topology shown in Fig. 1, the output tuples of task "`trans.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, are used to create 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 *`trans.9`* to *`gemm.6`* under "fields-grouping" is identified by the ***virtual mid*** where the target taskId *`gemm.6`* is replaced the alias "`gemm.1@`"

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

A *virtual mid*, such as `trans.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 its actual task-Id.

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

```
{trans.9^trmm.0@-30, trans.9^gemm.1@-35}
```

Indicating that the tuple is the 30th tuple sent from "`trans.9`" to a *trmm* task, and the 35th a *gemm* task. The recipient task with the recorded alias `trmm.0@`, can extract the matched *virtual-mid* `trans.9^trmm.0@-30` based on the *match of operation name* "`trmm`", 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 all other grouping types to direct grouping where 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, the combination of *mid-set* and *virtual mid* allows us to track the messaging channels of a task with multiple grouping criteria: 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.

5.4 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 Ts 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.

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.

6 Experiments

We have built the *Fontainebleau* platform and provided the failure recovery capability described 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, others hold workers daemons, each worker supervises several worker processes, and each worker process handles one or more tasks. The experiments are designed on the streaming process example shown in Fig. 1 with simulated data stream. However, in these experiments we focus on the performance of one task in the streaming process topology, because our goal is to check the *latency ratios* of (a) checkpointing versus non-checkpointing, (b) ASK-based versus ACK-based recovery, and (c) optimistic versus pessimistic checkpointing. In these cases the overall performance of multiple parallel tasks with overlapping disk-writes, etc., cannot give clear measures for the above ratios.

In the streaming process example shown in Fig. 1, the heaviest computation is conducted by tasks of operations "`gemm`" and "`trmm`" 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* 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. Our experiment results presented in this section is based on the *gemm* task.

6.1 Latency Overhead of Checkpointing

Let us first exam the impact of checkpointing to the performance of the streaming process involving GEMM operations. For this reason we focus on the performance ratio with and without checkpointing, particularly the turning point on the size of input matrices where checkpointing shows 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 1 K to 1 M, 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. 9 indicate that when the matrix dimension size N is smaller than 600, checkpointing has visible impact to the latency of the stream processing; after the matrix dimension size N overpasses 600, that impact becomes insignificant, since in that case the latency is dominated by the computation complexity.

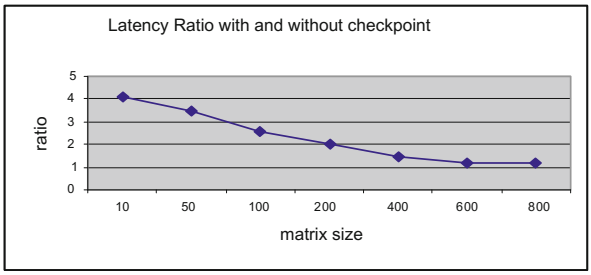

Fig. 9. Latency ration with and without checkpoint

6.2 ASK Versus ACK Based Recovery (Pessimistic)

In this experiment we compared the performance of the ASK-based transactional stream processing with the ACK based one, under pessimistic checkpointing. In our testing, the failure rate is set to 0.1 %. The matrix dimension size is fixed to 20. We measure the latency of a "*gemm*" task wrt its input tuples (a partition of tuples input to all "*gemm*" tasks) ranging from 1000 to 10000.

With the ACK based approach, a task does not move on to emit the next tuple until the success of processing the current tuple has been confirmed by the ACKs from all target tasks; otherwise the tuple will be re-sent after timeout. Such latency overhead is incurred during processing each tuple. However, under the proposed ASK based approach, a task does not wait for the acknowledgement to move forward as the acknowledgement is handled asynchronously to the task execution. In this case the corresponding latency overhead is only incurred during failure recovery which is rare. As a result, the ASK based approach can effectively improve the overall performance. Our comparison result shown in Fig. 10 has verified this observation.

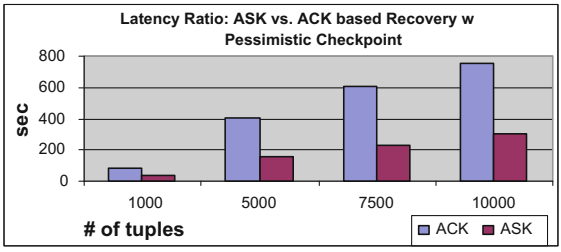

Fig. 10. Performance comparison of ASK and ACK based recovery mechanisms with pessimistic checkpointing

6.3 Optimistic Versus Pessimistic Checkpointing

In this experiment we compare the performance of optimistic and pessimistic checkpointing with the backtrack-based failure recovery mechanism. In our testing, the tuples are timestamped by seconds and the window boundary is set to 1 min with each containing approximately 100 tuples except the last window that contains less tuples;

the matrix dimension size is fixed to 20; and the failure rate is set to 0.1 %. The latency is measured on one "*gemm*" task with the input tuples (a partition of tuples input to all "*gemm*" tasks) ranging from 1000 to 10000. With pessimistic checkpointing, the output tuples must be persisted and acknowledged one by one (although asynchronous with execution), which incurs the performance penalty from both disk access and message delivery. With optimistic checkpointing, a task checkpoints, and is acknowledged, only once per window. Although the latency overhead incurred during failure recovery is higher, failures are rare thus the overall performance can be significantly enhanced. Our comparison result shown in Fig. 11 has verified this observation.

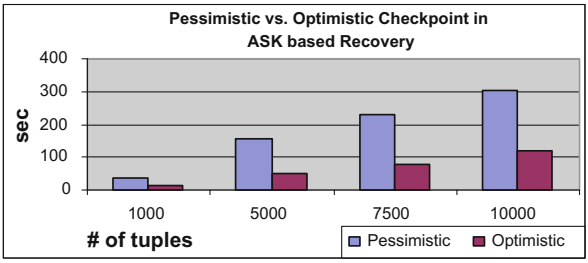

Fig. 11. Performance comparison of optimistic and pessimistic chepointing based recovery mechanisms

The performance comparison of optimistic and pessimistic checkpointing strongly depends on the computation complexity. The "g*emm*" tasks in our testing stream processing topology are computation-heavy compared with many event processing tasks thus the computation time contributes to a big portion of the elapse time, and the performance gain with the optimistic checkpoint mechanism is not very sharp. When the tasks are computation-lighter and the disk access cost for checkpoining is more dominated, the benefit of optimistic checkpointing becomes tremendous.

7 Conclusions

Fault-tolerance is the key requirement for applying stream processing to industry-scale and mission-critical applications; however the overhead for supporting fault-tolerance must be minimized to accommodate real-time analytics. This issue has been studied from various angles. The snapshot isolation model studied in the context of database transaction cares about stepwise state consistency but not failure recovery; the instant consistency of global state studied in the context of general distributed computing is too rigid for stream processing that is essentially based on eventual consistency. The checkpoint based failure handling is generally based on *pessimistic checkpoining* and *forward tracking*, i.e. a task checkpoints every output tuple before emitting, and waits for acknowledgement before rolling forward to emit the next one. The latency incurred by those approaches is too high to deal with real-time stream processing.

In this work we have taken an initial step to make transactional stream processing feasible to real-time stream processing. We integrated the *optimistic checkpointing*

mechanism with the *backtrack-based failure recovery* mechanism for the combined benefits. With the proposed *Continued stream processing with Window-based Checkpoint and Recovery* (CWCR) approach, we allow a task to checkpoint and acknowledge only once per window but continuously emit tuples in real-time stream processing. We also incorporated the inter-tasks checkpoint synchronization with the window semantics of stream processing, to eliminate the possibility of uncontrolled rollbacks. To implement these mechanisms on top of an existing stream processing platform where message routing is handled by separate system components inaccessible to individual tasks, we track physical messaging channels logically with the notions of *virtual channel*, *task alias* and *messageId-set*. To ensure the regular order of data flows not to be interrupted by the failure recovery process, we provided a task with the *designated messaging hub*, separated from the regular dataflow channel, for signaling ACK/ASK messages and for resending tuples,

We have implemented these mechanisms on *Fontainebleau*, the distributed stream analytics infrastructure we develop by extending the open sourced Storm platform. Our experiment results reveal the novelty of the proposed technologies, and most significantly, the feasibility to support fault-tolerance with minimized overhead for real-time stream processing.

References

1. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB J **15**(2), 121–142 (2006)
2. Abadi, D.J., et al.: The design of the Borealis stream processing engine. In: CIDR (2005)
3. Balazinska, M., Balakrishnan, H., Madden, S., Stonebraker, M.: Fault-Tolerance in the Borealis distributed stream processing system. In: SIGMOD 2005 (2005)
4. Botan, I., Fischer, P.M., Kossmann, D., Tatbu, N.: Transactional stream processing. In: EDBT 2012 (2012)
5. Johnson, D.B., Zwaenepoel, W.: Recovery in distributed systems using optimistic message logging and checkpointing. J. Algorithms **11**, 462–491 (1990)
6. Chen, Q., Hsu, M., Zeller, H.: Experience in continuous analytics as a service. In: EDBT 2011 (2011)
7. Chen, Q., Hsu, M.: Experience in extending query engine for continuous analytics. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DAWAK 2010. LNCS, vol. 6263, pp. 190–202. Springer, Heidelberg (2010)
8. Chen, Q., Hsu, M.: Query engine net for streaming analytics. In: Proceedings of 19th International Conference on Cooperative Information Systems (CoopIS) (2011)
9. DeWitt, D.J., Paulson, E., Robinson, E., Naughton, J., Royalty, J., Shankar, S., Krioukov, A.: Clustera: an integrated computation and data management system. In: VLDB 2008 (2008)
10. Franklin, M.J., et al.: Continuous analytics: rethinking query processing in a network-effect world. In: CIDR 2009 (2009)
11. Gedik, B., Andrade, H., Wu, K.-L., Yu, P.S., Doo, M.C.: SPADE: the system S declarative stream processing engine. In: ACM SIGMOD 2008 (2008)
12. Hwang, J.-H., Balazinska, M., et al.: High-availability algorithms for distributed stream processing. In: Proceedings of ICDE 2005, Washington, DC, USA (2005)

13. Johnson, D.B., Zwaenepoel, W.: Recovery in distributed systems using optimistic message logging and checkpointing. J. Algorithms **11**, 462–491 (1990)
14. Li, J., Karp, A.: Access control for the services oriented architecture. In: ACM Workshop on Secure Web Services (2007)
15. Shah, M.A., Hellerstein, J.M., Brewer, E.: Highly available, fault-tolerant, parallel datafows. In: Proceedings of SIGMOD, New York, USA (2004)
16. Prasad Sistla, A., Welch, J.L.: Efficient distributed recovery using message logging. In: Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (1989)
17. Stiegler, M., Li, J., Kambatla, K., Karp, A.: Clusterken: A Reliable Object-Based Messaging Framework to Support Data Center Processing, HPL-2011–44 (2011)
18. Tweeter, Transactional topologies (2012). https://github.com/nathanmarz/storm/wiki/Transactional-topologies
19. Wang, Y.M., Fuchs, W.K.: Optimistic message logging for independent checkpointing in message-passing systems. In: IEEE Symposium on Reliable Distribution System, pp. 147–154 (1992)