B. Mitschang et al. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017),
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 309

A Modular Approach for Non-Distributed Crash Recovery for Streaming Systems

Michael Brand[1] Marco Grawunder[2] H.-Jürgen Appelrath†[3]

Abstract: High availability and reliability are important aspects for streaming systems. State-of-the-art recovery techniques like active or passive standby use several nodes to fulfill these requirements. But not every processing task is done in a professional environment with resources for a distributed system (e.g. smart home). Additionally, even distributed streaming systems can benefit from reliable non-distributed crash recovery (NDCR) because it can help to restore the overall system state faster not only after a node failure but also after the roll-out of updates.

In this paper, we present our research on NDCR for streaming systems and point out its possibilities and limitations. We propose a flexible and extensible framework in which small NDCR tasks can be combined to high-level NDCR classes with different degrees of fulfillment for completeness and correctness: at-most-once, at-least-once or exactly-once. Further, we offer a way to mark elements that may be incorrect or duplicated.

Keywords: Streaming System, Non-Distributed, Availability, Crash Recovery, Framework, Odysseus

1 Motivation

If a streaming system is deployed, it should be robust and act comprehensibly even if system failures occur. Unfortunately, system failures lead to the loss of all data within the volatile main memory and streaming systems typically hold all data in the main memory. Mechanisms to recover from system failures are called crash recovery. Despite the fact that streaming systems hold their data in the main memory, there is another major difference to other data processing systems like DBMS. Streaming systems process data from active data sources that will not stop sending if the streaming system crashes. That makes data loss unavoidable without a backup of the incoming data streams. State-of-the-art systems use distributed solutions with recovery mechanisms like active or passive standby to achieve high availability. For an industrial or commercial use of streaming systems, this is a desirable solution, but streaming systems can also be used in non-industrial and non-commercial environments, e.g. in combination with the Internet of Things. An example is the monitoring and control of smart devices in a smart home. Private households often have no computer cluster for their streaming/smart home tasks and uploading the data in the cloud for data analysis is often also not favored because of privacy concerns. In such a scenario, there is a need for recovery mechanisms that at least reduce the data loss due to a system failure without having several streaming system nodes. Additionally, a system failure is not the

[1] Universität Oldenburg, Abteilung Informationssysteme, michael.brand@uni-oldenburg.de
[2] Universität Oldenburg, Abteilung Informationssysteme, marco.grawunder@uni-oldenburg.de
[3] Universität Oldenburg, Abteilung Informationssysteme

only scenario for recovery mechanisms. Another scenario is the roll-out of updates as it typically includes the restart of the streaming system. After a restart, the streaming system should automatically continue the stream processing tasks as soon as possible and reach the same state as if there were no processing gap.

Distributed streaming systems can also benefit from recovery mechanisms for stand-alone streaming systems. A crashed streaming system node that is recovered with stand-alone recovery mechanisms will be available again in less time to take part in a failsafe distributed stream processing. The recovered node can for example act as a backup for the former backup node that is now the active one. As for stand-alone streaming systems, it is also possible to use stand-alone recovery after the roll-out of updates. In order to keep the stream processing tasks running, the nodes in a distributed streaming system can be updated at different times. If a backup node gets updated, it should synchronize with the active node automatically and as soon as possible so that the active node can be updated as well. Additionally, the risk of data loss rises with the time needed to synchronize if there is only one backup node because the active node could crash during that time.

Despite state-of-the-art recovery works in a distributed way, we focus on recovery mechanisms for stand-alone streaming systems that we call *Non-Distributed Crash Recovery (NDCR)*. Such an NDCR covers at least the following, but typically even more, tasks in order to minimize the data loss:

1. Detect a system failure.
2. Reinstall all connections to data sources and sinks.
3. Reinstall and restart all queries.

There are different requirements for an NDCR depending on the application scenario. Some applications may tolerate data loss or duplicates, other may need full correctness of the results. To see if and how an NDCR can meet these different requirements, we adopt a classification for recovery in distributed streaming systems from Hwang et al. [H+05]. They distinguish between three different classes of recovery for distributed streaming systems. The weakest recovery class is called *gap recovery* and it ensures neither input nor output preservation. Therefore, a system failure results in missing results for all incoming elements before the streaming system can process them again (*at-most-once delivery*). Gap recovery addresses the needs of applications that operate solely on the most recent information (e.g. sensor-based environment monitoring) where dropping old data is tolerable for reduced recovery time and run-time overhead. Because elements may be aggregated, missing elements may also result in incorrect output values [H+05]. That can be a major disadvantage depending on the size and type of windows (types may be time window, e.g. the last 10 seconds, or element window, e.g. the last 10 elements).

Rollback recovery is the second recovery class and ensures that failures do not cause information loss. More specifically, it guarantees that the effects of all input elements are always forwarded to the instances that continue processing the elements despite failures (*at-least-once delivery*). Achieving this guarantee requires input preservation. Therefore, rollback recovery techniques result in a complete output stream (relative to its failure-free

counterpart), but typically they also result in duplicate values (all values between the last checkpoint and the system failure) [H+05]. The strongest recovery class, which is called *precise recovery*, completely masks failures and ensures that the output produced by an execution with failure and recovery is identical to the output produced by a failure-free execution (*exactly-once delivery*). Achieving this guarantee requires input preservation as well as output preservation. Every rollback recovery technique (e.g. passive standby) can fulfill the requirements for precise recovery if all duplicates that are a result of rollback recovery are eliminated [H+05].

Obviously, the possibilities of an NDCR to fulfill the requirements are not the same as for distributed streaming systems: all incoming elements that are sent by the data sources while the streaming system is not available must be processed or at least hold available by another entity to make completeness and correctness achievable. In a distributed streaming system, this can be done using another node but for non-distributed streaming systems other concepts are needed.

In this paper, we present our novel research on *NDCR for streaming systems* with the following contributions:

1. We adopt the recovery classification proposed by Hwang et al. [H+05] to non-distributed streaming systems.
2. We point out the possibilities and the limitations of an NDCR.
3. We propose a flexible and extensible framework in which small recovery tasks can be combined to high-level recovery techniques. By doing so, we allow new combinations to fulfill individual recovery requirements.
4. We propose a way to mark elements that may be incorrect due to recovery (wrong values after a gap or duplicates) because in our opinion it is crucial to make the subsequent faults of a system failure transparent for users and other systems. For example, results of an aggregation may be different (compared to a failure-free run) because of missing elements or duplicates.
5. Our evaluation shows the different costs in terms of latency and throughput for the respective recovery technique.

We implemented the recovery tasks in *Odysseus* [A+12], an open-source streaming system, as compositions of plug-ins that make the recovery component flexible and extensible. As a result of this flexible implementation, we can compose other recovery techniques than those with a minimum overhead (but maximum gap), full completeness or full correctness (e.g. a recovery technique that saves and restores operator states but does not preserve the input streams).

The remainder of this paper is structured as follows. In Section 2, we give a brief overview of related work. The following sections present the respective recovery classes for non-distributed streaming systems: gap recovery in Section 3, rollback recovery in Section 4 and precise recovery in Section 5. The setup and results of our evaluation are provided in Section 6. Finally, Section 7 concludes the paper and gives a brief overview of future work.

2 Related Work

Hwang et al. [H+05] discuss different requirements for recovery of a distributed streaming system and they propose three different classes of recovery techniques (cf. Section 1). *Borealis* [A+05] uses a gap recovery technique, called Amnesia, that restarts a failed query from an empty state and continues processing elements as they arrive. Additionally, it uses several rollback recovery techniques with extensions for precise recovery (e.g. passive standby in which each processing node periodically sends the delta of its state to a backup node that takes over from the latest checkpoint if the processing node fails).

Another gap recovery technique is proposed by Dudoladov et al. [D+15] for iterative data flows. It eliminates the need to checkpoint the intermediate state of an iterative algorithm in certain cases. In the case of a failure, Dudoladov et al. use a user-supplied so-called compensation function to transit the algorithm to a consistent state from which the execution can continue and successfully converge. This optimistic recovery has been implemented on top of Apache Flink[4] for demonstration.

OSIRIS-SE [BSS05] is a stream-enabled hyper database infrastructure with two main characteristics: dynamic peer-to-peer process execution where reliable local execution is possible without centralized control as well as data stream management and process management in a single infrastructure. Brettlecker et al. [BSS05] define reliability in the context of data stream processing: stream processes have to be executed in a way that the process specification is met even in the case of a failure. Further, they introduce a classification of failures with a class called *service failures* among others. A service failure indicates that at least one operator is no longer available. That includes system failures of individual nodes. For temporary failures, OSIRIS-SE uses buffering techniques while it uses backup nodes for permanent failures.

S-Store [M+15] is an extension of H-Store[5], an open-source, in-memory, distributed OLTP database system. It attempts to fuse OLTP and streaming applications. S-Store implements streams as well as windows as time-varying H-Store tables and uses triggers for a push-based processing. Because of that approach, Meehan et al. [M+15] declare stream states and window states in S-Store to be persistent and recoverable with H-Store mechanisms. But they do not point out what happens with data sent while S-Store is crashed due to a system failure. However, S-Store provides two recovery mechanisms: Strong recovery guarantees to produce exactly the same state as that was present before the failure. Weak recovery will produce a consistent state that could have existed but that is not necessarily the same state as that was present before the failure. In our understanding of a streaming system and a data stream itself, the state changes between failure and recovery due to new elements from active data sources.

Storm [T+14] and *Heron* [K+15] use Zookeeper[6] to track the execution progress and to recover the state of a failed node. Both systems use, like the others mentioned before, the possibilities of a distributed system for recovery.

[4] flink.apache.org
[5] hstore.cs.brown.edu/
[6] zookeeper.apache.org

Apache Kafka [KK15] is a publish-subscribe system where producers publish messages to topics and consumers read messages in a topic. For horizontal scalability, a topic can be divided into partitions. Brokers assign an offset (a monotonically increasing sequence number per partition) to each message and store the messages on disk. A consumer polls messages of a topic in a partition sequentially from a broker and tracks the offset of the last seen message. The offset is periodically checkpointed to stable storage to support recovery of consumers. If Kafka is used with multiple brokers, each partition is replicated across the brokers to tolerate broker failures.

3 Gap Recovery

Gap recovery techniques do not preserve incoming data streams. Therefore, the data that is sent by the data sources after the system failure is lost until the streaming system can continue the stream processing (*at-most-once delivery*) [H+05]. Section 3.1 analyses the impacts of gap recovery on the stream processing results and Section 3.2 describes the concept of our modular framework that fulfills the requirements for gap recovery.

3.1 Impacts on the Stream Processing Results

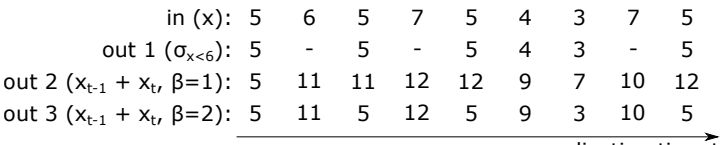
The length of a processing gap depends on four factors: the time needed to detect the failure, to correct the failure (especially for monolithic systems), to restart the streaming system (especially for monolithic systems), and the time needed for the recovery tasks (cf. Section 1). If the gap is measured in data stream elements (how many elements could the streaming system not process), the length of the gap depends also on the data rate of the incoming data streams [H+05]. For the scenario of an update (cf. Section 1), the first two factors are replaced by the time needed to update the streaming system.

Figure 1 illustrates the different possible impacts of a gap within incoming data streams. Figure 1a shows the incoming data stream *in* with simple numeric values. Three different output streams, *out* 1, *out* 2 and *out* 3, are the result of a simple filter operation and two aggregations respectively. *out* 1 contains all incoming elements that are less 6. *out* 2 sums each new input and the previous one up (sliding element window, slide $\beta = 1$). *out* 3 also sums the two elements up but without overlapping elements (tumbling element window, slide $\beta = 2$).

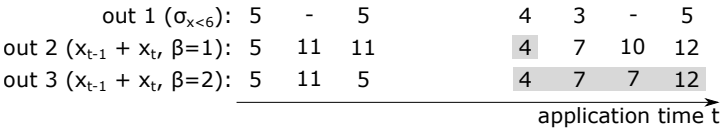
Figure 1b shows the same data stream processing but with a system failure causing a gap within the input stream. An obvious impact of the gap is a corresponding gap within the result streams, which we call *offline phase*.

Definition 1 (offline phase)**.**
The offline phase is the application time span between system failure and restart, in which elements are missing in the result stream and would be present in its failure-free counterpart.

For *out* 1, where each input element can be handled independently, the offline phase is the only impact. However, data stream operations often affect a set of data stream elements,



(a) Failure-free run.



(b) Run with system failure.

Fig. 1: A simple example for the different impacts of a gap within an incoming data stream.

e.g. relational aggregations like in *out* 2 and *out* 3 or joins. Typically, windows are used to define subsets of data stream elements that should be processed together, e.g. the average temperature of the last five minutes or the sum of the last two elements like in *out* 2 and *out* 3. Therefore, results that are not the same as their failure-free counterpart can occur after the offline phase. In Figure 1b, it is the first element after the offline phase in *out* 2. Here, the input value 4 and the previous one cannot be summed up because of the offline phase. Therefore, the result is different: 4 instead of 9. We call the period of results that may differ from their failure-free counterpart due to the offline phase *convergence phase*.

Definition 2 (convergence phase)**.**
The convergence phase is the application time span after an offline phase in which results exist but differ from results that would have been produced in the failure-free counterpart.

A gap within input streams can, therefore, cause a two-phase gap within result streams with an offline phase and a convergence phase. Whether a convergence phase exists depends on the operators and windows in the query plan as indicated by the example in Figure 1.

To distinguish different types of operators, we refer to the classification of Aurora operators that contains classes for arbitrary, deterministic, convergent-capable and repeatable operators [H+05]. Convergent-capable operators yield convergent recovery when they restart from an empty internal state and re-process the same input streams, starting from an arbitrary earlier point in time. Convergent recovery is a recovery that can result in a finite convergence phase after recovery (e.g. the sliding window in Figure 1). Hwang et al. [H+05] state that the window alignment is the only possible cause that prevents a deterministic operator from being convergent-capable (its window alignment does not converge to the same alignment when restarted from an arbitrary point in time). An example for an operator that is not convergent-capable is the aggregation with a tumbling window in Figure 1.

Convergent-capable operators are for their part *repeatable* if they yield repeating recovery when they restart from an empty internal state and re-process the same input streams, starting from an arbitrary earlier point in time [H+05]. Repeating recovery is a recovery that results in identical output elements after the offline phase compared to the failure-free

counterpart (no convergence phase). A necessary condition for an operator to be repeatable is to use at most one element from each input stream to produce a result (e.g. the filter in the example in Figure 1) [H+05]. The term "identical data streams" does not mean that every element must be at the same position in both data streams. The position can vary for elements with equal time stamps. Generally, we assume data streams to be ordered by time stamps.

Based on this operator classification, we conclude that a finite convergence phase exists if and only if the following constraints are fulfilled:

1. All operators are at least convergent-capable.
2. At least one operator is not repeatable.
3. The windows (failure run and failure-free run) do not contain the same elements.

Besides a finite convergence phase, there are two other options for a convergence: *immediate convergence* (no convergence phase) and *no convergence* (infinite convergence phase). Immediate convergence happens if all windows after the offline phase and their failure-free counterparts contain the same elements, or if all operators are repeatable and each element can be processed independently. For queries with operators that are not convergence-capable, the results may never converge. Non-determinism is one reason to have an infinite convergence phase (e.g. enrichment with randomized values). Another reason may be a shift of starting and ending points of windows because of the offline phase. An example is *out* 3 in Figure 1. It is the same sum operation like in *out* 2 but with tumbling windows with a slide of $\beta = 2$ instead of sliding windows with $\beta = 1$. If the first element after the offline phase is not a first summand in its failure-free counterpart, all sums after the offline phase will be calculated with other summands as their failure-free counterparts as in Figure 1b. Of course, it may also happen that there is no convergence phase after a system failure although not all operators are convergence-capable (e.g. non-deterministic behavior results in equal results by accident). Another aspect to mention is that not all data stream applications need a convergence of their results. A more detailed discussion about the types of convergences and the length of a convergence phase is outside the scope of this paper.

3.2 Modular implementation

The concept of our modular framework that fulfills the requirements for gap recovery provides several combinable subcomponents and is shown in Figure 2. Each subcomponent is responsible to back up particular information and to recover them if it is necessary. On the right, the persistent memory is illustrated. It is used by the subcomponents. Between the subcomponents and the persistent memory are different access layers integrated: *System Log*, *Processing Image* and *Backup of Data Streams (BaDaSt)*. A typical gap recovery technique consists of the five subcomponents at the top, *System State*, *Source and Sink Connections*, *Queries*, *Query States*, and *Window Alignments*, which will be explained below. The other subcomponents are needed for rollback and precise recovery. They will be explained in Section 4 and Section 5 respectively.

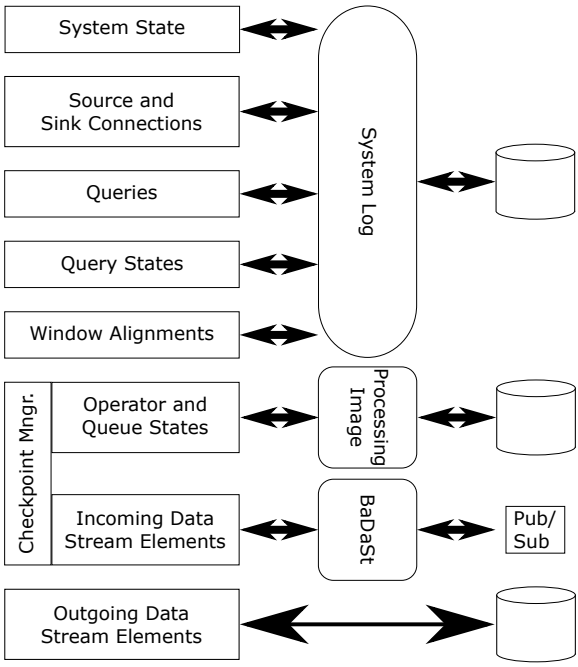


Fig. 2: Subcomponents of the NDCR to support various recovery techniques.

Before a streaming system can recover any information, it has to detect system failures. This can be done by logging all relevant system activities (e.g. the boot and shutdown processes). Therefore, we propose a *System Log* that is a protocol of rare system events that can be used to retrace relevant events. Additionally, it is possible to store other rarely changing events in the system log in order to recover them afterwards. The first subcomponent for gap recovery is the *System State* that has the following two functions:

1. Log successful boot and shutdown processes (write into system log).

2. While booting, search the system log for a missing shutdown event after the last boot event and notify the recovery component.

To fulfill the requirements of gap recovery (automatically continue processing after restart), it is necessary to take care of the connected data sources, the connected data sinks and the installed queries with their respective states. In this case, a query state means whether the query is running, paused, etc. All these aspects are typically rarely changing compared to data stream elements. This is why we propose to store this information in the system log. Therefore, the subcomponent *Queries* has the following three functions:

1. If a new query is installed, log the information that is necessary to reinstall it.

2. If a query is removed, log that information in order to prevent the installation of undesired queries.

3. While booting after a system failure (noticed by the subcomponent *System State*), search the system log for all queries that were installed after the last boot process and not removed afterwards and reinstall them.

The subcomponent *Source and Sink Connections* has similar functions. Closely related to the subcomponent *Queries* is the subcomponent *Query States* with the following functions:

1. If the state of a query change (e.g. paused, stopped, resumed), log that information in order to restore the latest query state.

2. For each reinstalled query (done by the subcomponent *Queries*), search the system log for the latest query state change and transform the query in the latest known state.

The best way to transform a query into its latest known state depends on the possibilities to switch from state to state and on the used streaming system. Intuitively, the functions of both subcomponents (*Queries* and *Query States*) could be done by a single subcomponent. We decided to separate them because reinstalling a query must be the first action for recovery of a query while recovering its state (e.g. start the query) must be done at the end. All other information to be recovered (e.g. operator states for other recovery techniques) should be done between reinstalling and restarting the query. Otherwise, the query processing would continue with empty operator states instead of recovered ones. Note in this context that gap recovery does not require the backup of operator states. Because of the offline phase, all operator states that were up-to-date before a system failure are typically outdated after system recovery (because of the progress in the incoming data streams).

The last subcomponent for gap recovery is called *Window Alignments*. As mentioned above, windows that do not contain the same data stream elements as their failure-free counterparts can cause a convergence phase. To limit the number of those "corrupted" windows (important for a finite convergence phase), there have to be windows that start after the offline phase and that contain the same data stream elements as their failure-free counterparts. To achieve that, all windows after recovery must have starting points that are also starting points in the failure-free run. For application-time windows with a constant slide (e.g. β *seconds*), it can be achieved by logging the starting point of the first window. All starting points after recovery can then be calculated using the logged starting point. Note that fixing the window alignment does typically not work for system-time windows and element windows. The content of a system-time window depends on execution-specific properties (e.g. how fast data is read). For element windows, the number of elements inside the offline phase is unknown if the data stream does not contain any counting information. Therefore, we only consider application-time windows for the subcomponent *Window Alignments*, but we intend to find solutions for element windows, too. The subcomponent *Window Alignments* has the following functions:

1. If a query with application-time starts processing, log the time stamp that is the starting point of the first window.

2. After reinstalling a query that uses application-time (done by the query state subcomponent), the starting points of the windows to come must be aligned with the logged starting point.

Consider the example in Figure 1b to clarify the functions of the subcomponent *Window Alignments*. The data source has a constant data rate of one element per second. *out* 3 uses application-time windows with a width of $\omega = 2\ seconds$ and a slide of $\beta = 2\ seconds$. The query starts and the window operator receives the first element with a time stamp ts_0. This means that all time stamps ts_i are starting points of windows for which $(ts_i - ts_0) \bmod 2\ seconds = 0$ is true. Without logging and recovering ts_0, the time stamp of the first element after the system restart would be interpreted as ts_0. But logging and recovering ts_0 results in correct starting points. Note in this context that the subcomponent *Window Alignments* is not necessary for every streaming system. If the time windows in a failure-free run are already aligned by using a constant point in time (e.g. $ts_0 = 0$), the alignment remains the same after a system failure.

Another important aspect for gap recovery is *syntactic transparency* that we define similar to Gulisano et al. [G+12] who define it for parallelization of queries.

Definition 3 (syntactic transparency)**.**
Syntactic transparency means that query recovery should be obvious to the user in a way that the user can identify results that may be not correct due to recovery.

To achieve syntactic transparency, we propose to annotate results that are within a convergence phase because they might not be equal to their failure-free counterparts. As we explained in this section, the convergence phase depends on the used operators and windows (we consider either some kind of time windows or element windows). The annotation that indicates whether an element is inside a convergence phase or not is defined as follows:

Definition 4 (trust)**.**
The annotation trust for an element is one of the three following values: `trustworthy`, which is the default value, `untrustworthy` for elements that are incorrect due to recovery and `indefinite` if it cannot be determined whether an element is incorrect due to recovery.

The trust annotation is a meta attribute of data stream elements that makes it necessary to merge several trust values if the corresponding elements are merged (e.g. in an aggregation or join). To maintain syntactic transparency, we implemented the merge function for the trust annotation as a min-function: if an element gets aggregated or joined with another element with a lower trust, its trust will be decreased to that lower trust. Using a merge function for the trust values allows for application-time windows to set only the trust of the first element after a system restart to `untrustworthy`. All results that are based on the first element after restart get the same reduced trust level. Results that are not based on the first element but on younger ones are after the convergence phase and their trust is not decreased. For element windows, it cannot be determined whether an element is inside a convergence phase or not. This is because the information about the number of elements that are in an offline phase (those we missed) is typically not available. Therefore, we propose to set the trust value to `indefinite` for all elements after gap recovery if element windows are used.

4 Rollback Recovery

Rollback recovery techniques can be used to avoid information loss because they guarantee completeness [H+05]. Section 4.1 analyses the impacts of rollback recovery on the stream processing results and Section 4.2 describes the components of our modular framework that are needed to fulfill the requirements for rollback recovery.

4.1 Impacts on the Stream Processing Results

We define *completeness* as follows:

Definition 5 (completeness)**.**
A complete result after recovery of a streaming operation contains all elements of its failure-free counterpart and no elements that its failure-free counterpart does not contain.

This requires input preservation, but the result streams after rollback recovery can also contain duplicates (*at-least-once delivery*) [H+05]. Hwang et al. [H+05] discuss rollback recovery for distributed streaming systems and we propose to assign their basic idea of rollback recovery to non-distributed streaming systems.

To achieve completeness, all elements that could not be processed due to a system failure have to be available after system recovery in order to process them. We call an application that is responsible for the backup of the incoming data streams *Backup of Data Streams (BaDaSt)*. Generally, there are two ways to put BaDaSt into practice offering some limitations for non-distributed streaming systems:

- Local: BaDaSt and the streaming system share resources (e.g. running on the same machine).
- Distributed: BaDaSt and the streaming system share no resources (they communicate via network).

The advantage of the distributed solution is the higher guarantee of completeness respectively the lower risk that a system failure impacts both streaming system and BaDaSt. Less communication costs and less hardware that has to be assigned to the data stream processing task are the advantages of the local solution. Besides these two ways of where to run BaDaSt, there are also different possibilities what application to use: specialized in-house development, publish/subscribe system, another streaming system, etc. We recommend to use the publish/subscribe system Apache Kafka [KK15] because it has already demonstrated that it can be a good solution to log data streams in a fault-tolerant and partitioned way [KK15]. For a local solution, a single Kafka broker can be used on the same machine, whereas a cluster of brokers on different machines can be set up for a distributed solution.

In order to reduce recovery time, it is not suitable to log all elements that have ever been received. As for database management systems [ÖV11], checkpoints should be used to limit the length of the logs.

As mentioned before, rollback recovery may result in duplicates. The elements that are duplicates are all elements that are logged by BaDaSt and have successfully been processed by the streaming system before a system failure occurred (all elements between the last checkpoint and the system failure). Whether the usage of BaDaSt is sufficient for rollback recovery, depends (as the convergence phase in Section 3) on the used operators and windows.

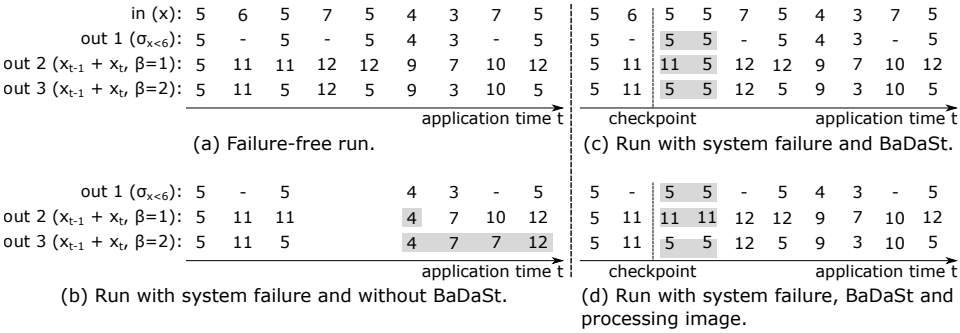


Fig. 3: The simple example (cf. Figure 1) enhanced for the impacts of BaDaSt and processing images.

Consider the same example as in Section 3 that is shown again in Figure 3a and Figure 3b. *in* is the incoming data stream, *out* 1 is the result of an operation that selects all elements lower than 6, and *out* 2 is the result of an operation that sums the current element and the last one up (sliding window). *out* 3 also sums the two elements up but without overlapping elements (tumbling window). Figure 3c illustrates the impacts of BaDaSt where all elements of *in* after a given checkpoint are logged. As one can see, the third element of *in* (5) appears twice because it is the only element that is between the checkpoint and the system failure (the gap). Since *out* 1 is the result of a repeatable operation, the third value of *out* 1 (5) is duplicated. *out* 2 is convergence-capable and it converges after one element because of the overlapping element windows (the current value is added to the last value). This results in the "wrong" value of 5, which would be 11 without system failure (cf. the third and fourth elements of *out* 2 that should be duplicates). *out* 3 is deterministic, but its third value (5) is correctly duplicated because it is the first element of the current tumbling window. To avoid convergence phases and therefore to reach completeness, a processing image is additionally needed for deterministic and convergence-capable operators.

Definition 6 (processing image)**.**
A processing image contains all operator and queue states in a query processing of a streaming system at a given point in time. An operator state consists of all information an operator stores for the next element to process (e.g. sums, counts). A queue state consists of all elements that are in the streaming system waiting to be processed by an operator (we assume streaming operators to be connected with queues as in [GÖ03]).

The points in time at which processing images are written should be the same as those at which the logs of BaDaSt start: the checkpoints. Figure 3d illustrates the impacts of a processing image together with BaDaSt. The processing image is written at the given checkpoint and all elements of *in* that have a time stamp after the checkpoint are logged. The

difference to Figure 3c is that the fourth element of *out* 2 is a duplicate because the partial sum (6) was stored as operator state. Therefore *out* 2 is complete (cf. *out* 2 in Figure 3a).

4.2 Modular implementation

To fulfill the requirements of rollback recovery, we propose several subcomponents to be built on top of the subcomponents for gap recovery. The new subcomponents are *Checkpoint Manager*, *Operator and Queue States*, and *Incoming Data Stream Elements* (cf. Figure 2). They use the already discussed *Processing Image* and *BaDaSt* as access layer. The *Checkpoint Manager* is a special subcomponent because it does not backup or recover information. It manages the checkpoints for the given stream processing task and notifies all interested entities (listeners) when a checkpoint is reached. The implementation of the *Checkpoint Manager* is easily exchangeable to allow different strategies (e.g. set a checkpoint every n elements, every n application time instants, or every n system time instants). To avoid concurrent modifications, the query processing has to be suspended if a checkpoint is reached. If the processing would continue while for example the processing image is stored, a correct image could not be guaranteed.

The processing image is stored and recovered by the subcomponent *Operator and Queue States* with the following functions:

1. If a checkpoint is reached, write for each stateful operator and queue its state in the processing image.
2. After reinstalling a query, load for each stateful operator and queue of that query its state from the processing image.

The second new subcomponent for rollback recovery handles backup and recovery of the *Incoming Data Stream Elements*:

1. If a new data source, whose data stream shall be logged, is connected, connect BaDaSt to that data source. From there on, BaDaSt stores all elements it receives.
2. If a checkpoint is reached, the streaming system determines the last received element of a data source and stores that information persistently (e.g. as an operator state).
3. After reinstalling a query, the streaming system connects to BaDaSt and sends the last element that has been received before the system failure. BaDaSt answers with all elements (in temporal order) that it has received after that element.

We propose to integrate a new *Source Recovery* operator in a query for the second and third function. This operator has the original data access operator and BaDaSt as inputs. It needs the element from the last checkpoint to recover after a system failure (cf. second function). First, it sends that element to BaDaSt in order to receive all stored elements that are in the temporal order after that one. From now on, the operator receives two input streams, one from the original data source and one from BaDaSt. To decide which element should be sent to the other streaming operators, the source recovery operator has three modes: `BaDaSt`, `Switch`, and `Source`. `BaDaSt` indicates that the elements from BaDaSt are older than the

elements from the data source (the current element from source is the watermark). If that is the case, the elements from BaDaSt will be transferred and the elements from the data source will be discarded. The operator switches to `Switch` if the elements from BaDaSt are not longer older as the elements from the data source. The last transferred element from BaDaSt becomes the watermark. `Switch` remains until we also receive the watermark from the data source. From this point in time on, we can transfer all subsequent elements from the data source without data loss (`Source`). Additionally, the *Source Recovery* operator can annotate all elements that are before the first element from the data source with a lower trust (`indefinite`, cf. Section 3) to indicate that they might be duplicates. Note that only those elements are duplicates that are received between the last checkpoint before the system failure and the system failure. All elements within the offline phase are no duplicates. But the option to annotate only the elements that are indeed duplicates is in this case not sufficient: if a duplicate is identified, it can be removed. That would be no more rollback but precise recovery.

As a conclusion of this section, it is also possible to deduce further recovery techniques with partial completeness guarantees (no 100% guaranteed completeness for all deterministic operations). An example of such a recovery technique has already been illustrated in Figure 3c. Here, the usage of BaDaSt without a processing image was sufficient for *out* 1 and *out* 3. This *stateless rollback recovery* is first of all useful for queries that consist solely of repeatable operations (*out* 1) because completeness can be guaranteed. Secondly, if tumbling (non-overlapping) time windows are used (*out* 3), the checkpoints can be set between consecutive windows. Then completeness can also be guaranteed. For other deterministic operations, the result is a convergence phase until all elements of a complete window are after the checkpoint. All later results that are calculated before the system failure are duplicates (*out* 2).

5 Precise Recovery

In contrast to gap or rollback recovery techniques, precise recovery techniques shall completely mask all failures and ensure semantic transparency (*completeness*, *exactly-once delivery*) [H+05]. In this context, we define *semantic transparency* closely linked to a definition by Gulisano et al. [G+12] for parallelization of queries:

Definition 7 (semantic transparency)**.**
Semantic transparency means that, for a given input, recovered queries should produce exactly the same output as their failure-free counterparts with the exception that, for elements with same time stamps, the order can differ.

Achieving this guarantee requires input preservation as well as output preservation. The only impact for the user is a temporary slower processing [H+05]. Hwang et al. [H+05] state that every rollback recovery technique can fulfill the requirements for precise recovery if all duplicates, which are the result of rollback recovery, are eliminated.

To eliminate duplicates, the recovery component needs to know those elements that the streaming system already sent to the data sinks (those that indeed left the system). One

possibility is to backup all outgoing data stream elements similar to the backup of the incoming data stream elements (cf. Section 4). Another more suitable solution is to track only the progress of a result stream. This is done by storing only the last sent element persistently (and override the older one every time). After system failure and rollback recovery techniques, the stored element can be used to identify all duplicates that are all elements before that element (incl. the element itself). For elements that are intended to be duplicates, additional information have to be logged in order to differentiate between them. This can be a simple counter for equal elements.

The new subcomponent *Outgoing Data Stream Elements* (cf. Figure 2) uses the persistent memory directly as data structure and has the following functions:

1. If a data stream element is sent by the streaming system as a result to a data sink, the information needed to identify that element is stored persistently in a file.
2. After reinstalling a query, the last stored information will be loaded from file. Before sending any new results, the results will be compared with that watermark and only newer elements will be sent.

With that extension to rollback recovery (cf. Section 4), the requirement for precise recovery (correctness) is fulfilled.

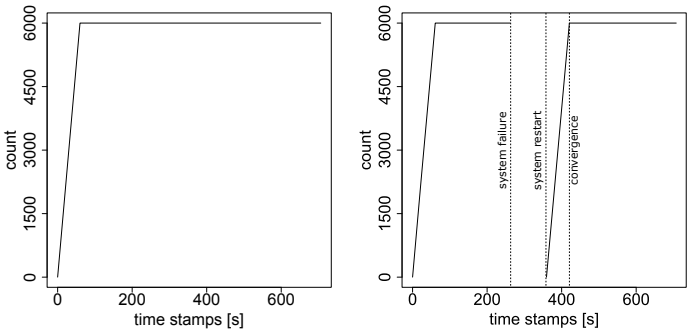
6 Evaluation

In this section, we present our evaluation results. Section 6.1 explains the evaluation setup, thus the used streaming system and query. In Section 6.2, we will show the completeness and correctness results for each recovery class and in Section 6.3 the syntactic transparency of our recovery architecture. Finally, we present the latency and throughput results in Section 6.4.

6.1 Setup

As mentioned before, we implemented our crash recovery architecture in *Odysseus* [A+12]. For our evaluation, we decided to choose a scenario that is easy to understand and to reproduce. The used data source sends elements with strictly monotonously rising time stamps every 10 ms. We installed a query in Odysseus [A+12] that counts the elements in sliding application-time windows with a width of $\omega = 60\ s$ and a slide of $\beta = 10\ ms$.

The expected result of a failure-free run is shown in Figure 4a. After starting the query, the count increases until the first window is complete (resulting in 6000 elements in a window for the used data rate). Afterwards, the count is constant due to the fact that for every new window, a new element is added and the oldest one is removed (slide of 10 ms matches the data rate). For the experiments with the recovery techniques, we let Odysseus [A+12] crash after 270 $seconds$ and we restarted it 90 $seconds$ later (both system-time). The checkpoints were set every 60 $seconds$ (system-time) and we used Apache Kafka [KK15] with a single, local broker for BaDaSt. Each experiment was repeated 10 times to get reasonable results.

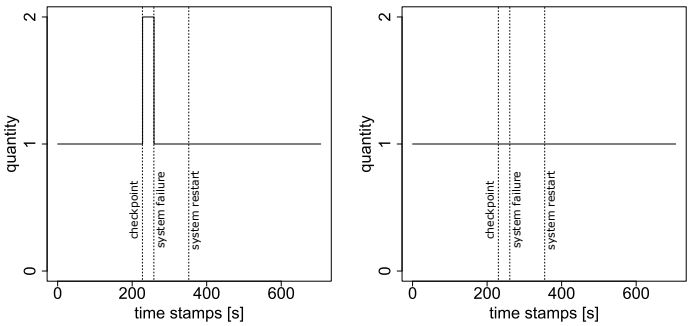


(a) Failure-free run. (b) Gap Recovery.

Fig. 4: The results of the counting aggregation.

6.2 Completeness and Correctness

In this subsection, we will analyze the completeness and correctness of the results of the counting aggregation. Figure 4b presents the results of the experiments with gap recovery. Before the system failure, all counts are the same as in the failure-free run. The offline phase lasts until the system restarts and is characterized by missing counts. Since no operator states (e.g. counts) are recovered, the count starts again with 0 after the offline phase and the results converge when the count is 6000 again. We omit to show the result stream for the experiments with rollback recovery respectively precise recovery. They are equivalent to the results of the failure-free counterpart because of their completeness (cf. Figure 4a).



(a) Rollback Recovery. (b) Precise Recovery.

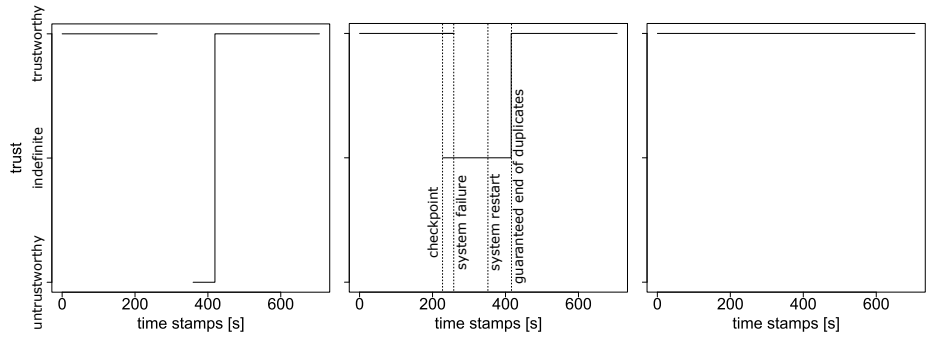
Fig. 5: The quantity of timestamps in the result stream.

To visualize the duplicate phase for rollback recovery, Figure 5a shows the quantity of each time stamp in the result stream for rollback recovery. In a failure-free run, each time stamp would occur exactly once, and in Figure 5a, all time stamps occur at least once (completeness), but there is a time span in which time stamps occur twice in the result streams: the duplicate phase between the last checkpoint and the system failure. Figure 5b

shows the quantity of the time stamps for precise recovery. Since all duplicates are eliminated, each time stamp occurs once (correctness).

6.3 Syntactic Transparency

As described in Section 3, we achieve syntactic transparency for our recovery architecture by annotating result streams with a trust value (`trustworthy`, `untrustworthy` or `indefinite`). Figure 6 shows the trust results for our experiments.



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

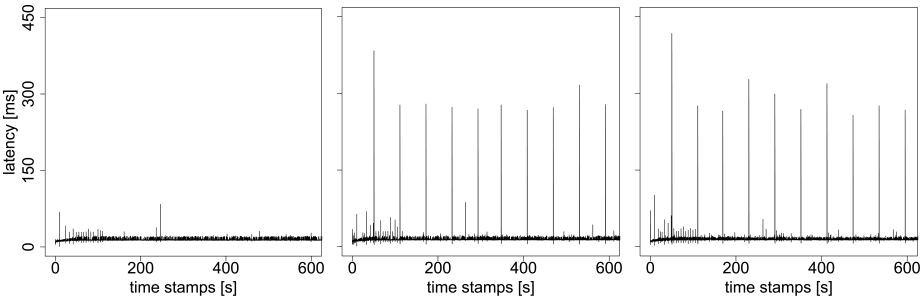
Fig. 6: The syntactic transparency for the results of our evaluation.

Figure 6a visualizes the trust for the gap recovery experiment. As in Figure 4b, the offline phase is evident. For the subsequent convergence phase, the trust is decreased to `untrustworthy` because the calculated counts are incorrect. Afterwards, the results are trustworthy again. Figure 6b shows the trust for the rollback recovery experiments. There is no gap within the trust values, but there are time stamps with two trust values. Those are the duplicates, each `trustworthy` before the system failure and `indefinite` after recovery (cf. Figure 5a). The trust is `indefinite` until the recovery component can guarantee that there are no more duplicates. This is when the streaming system receives the first element from the data source after system restart also from BaDaSt. The results in case of precise recovery shown in Figure 6c are always `trustworthy` (correctness).

6.4 Latency and Throughput

In the experiments for the latencies and throughputs, we used the backup mechanisms of the respective recovery class but without creating a system failure. Therefore, the experiments show the costs of the respective recovery class for its backup mechanisms.

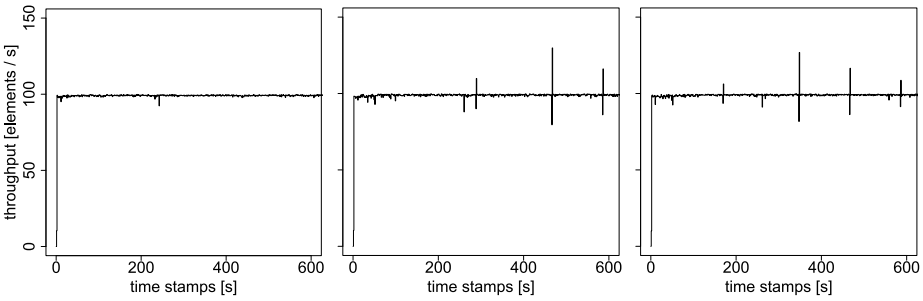
Figure 7 presents the latencies as the time between receiving the input element that triggered the output and sending the result. The latencies for the gap recovery experiments in Figure 7a and those of experiments without any backup mechanism are indistinguishable (we omit to show the latter). This is because there is no backup of information at the run-time of a query



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

Fig. 7: The latency results of our evaluation.

if gap recovery is used (cf. Section 3). The median is approx. 13.33 *ms* and the average approx. 13.62 *ms*. For rollback recovery in Figure 7b, the latencies have peaks resulting in a higher average of 15.07 *ms*, but the median is almost the same (approx. 13.47 *ms*). The peaks match the checkpoint interval of 60 *s* (system-time) and the latencies increase because of the actions that are done: suspend processing, store processing image, and resume processing (cf. Section 4). The storing of each outgoing element is the difference between rollback recovery and precise recovery in Figure 7c, but with little effect on the latencies. The median is approx. 13.58 *ms* and the average approx. 14.97 *ms*.



(a) Gap Recovery. (b) Rollback Recovery. (c) Precise Recovery.

Fig. 8: The throughput results of our evaluation.

Figure 8 shows the throughputs as the number of incoming elements that can be processed per second. The impact of the backup mechanisms is low and the medians are approx. the following: 99.15 $elements/s$ for gap recovery in Figure 8a, 99.07 $elements/s$ for rollback recovery in Figure 8b and 99.06 $elements/s$ for precise recovery in Figure 8c. The theoretical maximum is 100 $elements/s$ (cf. data source in Subsection 6.1). The peaks (positive and negative) for rollback and precise recovery are caused by the suspension and resuming of the query at each checkpoint.

We evaluated the same query with a data source sending an element every millisecond. Due to place limitations we omit to show the results, but the costs for rollback and precise recovery increase dramatically (median up to 50 *s*). This is because Odysseus has to buffer much more elements if a checkpoint is reached. For such a scenario, a gap or a distributed recovery should be used.

7 Conclusion & Future Work

In this paper, we presented our flexible and extensible framework for an *NDCR for streaming systems*. We adopted a classification of recovery techniques for distributed streaming systems that contains three classes: gap recovery (at-most-once delivery), rollback recovery (at-least-once delivery) and precise recovery (exactly-once delivery). Since the results of gap recovery contain an offline phase and (dependent on operators and windows) a convergence phase, we calculate the length of a convergence phase based on application-time windows and annotate elements that are inside the convergence phase. That makes potentially incorrect results transparent to the user.

The limitations of NDCR are obvious because incoming elements have to be stored to achieve completeness (rollback recovery) and it has to be done outside the streaming system. One has to compromise about reliability and costs because the possibility increases that both streaming system and backup of data streams crash due to the same system failure with the number of shared resources. Additionally, processing images are needed in many cases for completeness. Similar to the elements inside a convergence phase, we annotate duplicates that are a result of rollback recovery with a decreased trust value. With an elimination of duplicates on top of rollback recovery, we achieve correctness (precise recovery).

We implemented the described recovery framework in the open-source streaming system *Odysseus* [A+12] and our evaluation results show the following conclusions:

1. The combinations of NDCR subcomponents to recovery techniques fulfill the respective requirements (completeness, correctness, syntactic transparency).
2. There are no run-time costs for gap recovery.
3. For rollback recovery, the costs in terms of increased latencies are only given for elements that are inside the streaming system while a checkpoint is reached.
4. It is not worth to use rollback recovery because the additional costs for precise recovery are low.
5. Rollback or precise recovery is only feasible for incoming data streams with moderate data rates or checkpoints in very small intervals.

The third point indicates that one has to compromise about backup overhead and recovery time when at least rollback recovery is used.

For future work, we intend to analyze the correlation between the used operators and windows on the one hand and the type of convergence phase (immediate convergence, finite convergence, no convergence) and its length on the other hand in more detail. For

element windows, we also intend to find a solution to avoid infinite convergence phases. Those can occur if the starting points of the element windows are not the same as for their failure-free counterparts. Finally, we plan to evaluate other recovery techniques such as *stateless rollback recovery* that should decrease the backup overhead.

References

[A+05] Ahmad, Y.; Berg, B.; Cetintemel, U., et al.: Distributed Operation in the Borealis Stream Processing Engine. In: Proceedings of the 2005 ACM SIGMOD. ACM, pp. 882–884, 2005.

[A+12] Appelrath, H.-J.; Geesen, D.; Grawunder, M., et al.: Odysseus – A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems. ACM, pp. 367–368, 2012.

[BSS05] Brettlecker, G.; Schuldt, H.; Schek, H.-J.: Towards Reliable Data Stream Processing with OSIRIS-SE. In: Datenbanksysteme in Business, Technologie und Web (BTW) 2005. Pp. 405–414, 2005.

[D+15] Dudoladov, S.; Xu, C.; Schelter, S., et al.: Optimistic Recovery for Iterative Dataflows in Action. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1439–1443, 2015.

[G+12] Gulisano, V.; Jimenez-Peris, R.; Patino-Martinez, M., et al.: StreamCloud: An Elastic and Scalable Data Streaming System. Parallel and Distributed Systems, IEEE Transactions on 23/12, pp. 2351–2365, 2012.

[GÖ03] Golab, L.; Özsu, M. T.: Issues in Data Stream Management. ACM Sigmod Record 32/2, pp. 5–14, 2003.

[H+05] Hwang, J.-H.; Balazinska, M.; Rasin, A., et al.: High-Availability Algorithms for Distributed Stream Processing. In: Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on. IEEE, pp. 779–790, 2005.

[K+15] Kulkarni, S.; Bhagat, N.; Fu, M., et al.: Twitter Heron: Stream Processing at Scale. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 239–250, 2015.

[KK15] Kleppmann, M.; Kreps, J.: Kafka, Samza and the Unix Philosophy of Distributed Data. IEEE Data Engineering Bulletin/, pp. 4–14, 2015.

[M+15] Meehan, J.; Tatbul, N.; Zdonik, S., et al.: S-Store: Streaming Meets Transaction Processing. In. 2015.

[ÖV11] Özsu, M. T.; Valduriez, P.: Principles of Distributed Database Systems. Springer Science & Business Media, 2011.

[T+14] Toshniwal, A.; Taneja, S.; Shukla, A., et al.: Storm @Twitter. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, pp. 147–156, 2014.