Pedro Miguel Fortunato Silvestre

Degree in Computer Science and Engineering

Consistent High-Availability for Distributed Streaming Computations

Dissertation plan submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: João Carlos Antunes Leitão, Assistant Professor,
NOVA University of Lisbon

Co-adviser: Asterios Katsifodimos, Assistant Professor,
Delft University of Technology

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

February, 2019

The work reported in this document was partially supported by
Fundação para a Ciência e Tecnologia through the Project NG-STORAGE (PTDC/CCI-INF/32038/2017)

Abstract

Distributed dataflow computations have emerged in response to the need for processing large datasets quickly. Stream processing, a type of distributed dataflow, has enabled real-time analytics, with low latency and high throughput. A recent trend is their use for building reactive systems, which require high-availability and additional processing or delivery guarantees. Previous work has provided high-availability at the cost of losing consistency in the dataflow graph, due to the inherent nondeterminism present in dataflow computations. The state-of-the-art approach to fault-tolerance is the use distributed snapshotting to be able to rollback the dataflow graph to a consistent state. This approach has two issues. First, recovery is global, coordinated, and slow, meaning that use-cases requiring high-availability cannot rely on it. Second, in order to provide exactly-once delivery, the system must wait for a checkpoint to complete before being allowed to output, increasing latency. In this thesis, the design space of the fault-tolerance techniques of distributed dataflow systems is explored.

Subsequently, a novel approach for recovering stateful operators using causal logging is devised, filling an open space in the set of offered approaches. Unused computational power in the system is used to maintain passive stand-by operators, which can quickly take over execution of a failed operator. Causal logging is employed to bring that operator to a consistent state with low overhead. This approach promises to deliver faster recovery times by recovering only the failed operators. Since it does not require global coordination, unconnected components will be able to make progress, even under failure. This approach is configurable, allowing for a trade-off between safety and metadata overhead. The use of causal logging also reveals new opportunities for dataflow computations, such as low latency transactionless exactly-once delivery.

A prototype shall be implemented on Apache Flink and evaluated experimentally on its impact to throughput and latency, under different consistency guarantees and failure scenarios.

Keywords: stream processing, dataflow, fault-tolerance, exactly-once, causal logging

Resumo

Computações em dataflow distribuído emergiram em resposta à necessidade de processar grandes conjuntos de dados rápidamente. Processamento de streams, um tipo de computação em dataflow distribuído, permite análise de dados em tempo real, com baixa latência e alto débito. Uma tendência recente é o seu uso para criar sistemas reativos, que necessitam de alta disponibilidade e adicionais garantias de processamento ou entrega. Trabalho prévio implementa alta disponibilidade à custa de consistência no grafo de computação, devido ao não determinismo inerente em computações de dataflow. A abordagem à tolerância de falhas do estado-da-arte é o uso de snapshots distribuídas, que permite reverter o grafo computacional a um estado consistente. Esta abordagem tem dois problemas. Primeiro, a recuperação é global, coordenada e lenta, portanto casos de uso necessitando de alta disponibilidade não a podem usar. Segundo, para oferecer garantias de entrega exactamente-uma-vez, o sistema tem de esperar que um checkpoint complete para poder entregar resultados, aumentando a latência. Nesta tese, o espaço de desenho de técnicas de tolerância a falhas para sistemas de dataflow distribuído é explorado.

Subsequentemente, uma nova abordagem para recuperar operadores com estado usando causal logging é desenhada, preenchendo um espaço em aberto no conjunto de soluções existentes. Poder computacional em excesso é usado para manter operadores stand-by, que podem rápidamente assumir o controle da execução de um operador falhado. Causal logging é usado para trazer um operador a um estado consistente com baixo custo. Esta abordagem promete recuperação rápida ao recuperar apenas os operadores falhados. Dado que não requer coordenação global, componentes desconexos continuam a progredir, mesmo durante falhas. Esta abordagem é configurável, permitindo uma troca entre segurança e custo de metadados. O uso de causal logging revela novas oportunidades, como entrega exactamente-uma-vez de baixa latência sem usar transações.

Um protótipo será implementado no Apache Flink e avaliado no seu impacto ao débito e latência, sobre diferentes garantias de consistência e cenários de falha.

Palavras-chave: processamento de streams, dataflow, tolerância a falhas, causal logging

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

The recent interest in deriving valuable insights from data has caused enterprises to start storing massive amounts of data. The *Internet of Things* movement further increased this, by turning normal devices into data producing smart devices. The combination of these factors sparked the development of distributed dataflow systems, such as MapReduce[25] and later Apache Spark[60]. Classically, these systems are identified by their data-parallel programming model, where programs are built as flows of data between deterministic side-effect free operators, which allows these jobs to be expressed as Directed Acyclic Graphs[35] (DAGs). Two other important factors that led to the popularity of these systems are their transparent recovery mechanisms, alleviating users from implementing complex fault-tolerance logic, and also automatic scheduling of tasks with data locality awareness, leading to high throughput.

The continued increase of data sources means that we are producing more data than is convenient to store, leading to the appearance of *data swamps*, enormous pools of raw data with no use due to lack of organization[18]. This combined with new use-cases requiring *fast data* has led to the advent of *Stream Processing*[24], where analytics can be performed on streams of data in real-time, thus removing the need to store the data after deriving the insights. First, stateless systems were developed, but soon after the need for stateful stream processing became apparent, leading to the success of platforms such as Apache Flink[21].

The volumes of data being continuously processed keep increasing and to keep up, deployments have scaled to thousands of nodes and enormous state [19]. However, as the number of participants in a computation increases, so does the probability that at any given moment, one of them may fail. Modern approaches to recovery in dataflow systems rely on rolling back the state of the computation to a previous state, sometimes reffered to as a *stop-the-world* approach[62], since all participating processes must pause for recovery. The full graph must be rolled back, because of the nondeterminism present in dataflow computations, such as message delivery order. In the presence of common failures or dynamic environments, where operators are moved from node to node this approach is not able to make progress[62]. This limits their usability in both Edge computing scenarios as well as popular on-demand “Spot Instances”.

Fast data use-cases, such as online fraud detection, additionally require high availability in order comply with strict SLAs. Current stream processing systems are unable to provide both high availability and exactly-once delivery guarantees, while maintaing their original promise of low latency. On the other hand, interest in the use of stream processors to implement arbitrary services as reactive systems has resulted in the implementation of several prototypes[2, 26, 42, 49]. These systems inherently require exactly-once processing and delivery, and also high-availability, as latency, and by extension downtime, translates to a direct loss in revenue[36].

Whether a system offers the capabilities described above is dependent on their approach to fault-tolerance. Currently, no comprehensive analysis of the design-space of the fault-tolerance of distributed dataflows exists, meaning users and developers must independently discover the inherent trade-offs. Perhaps due to this, an alternative has not yet been explored, *causal logging*, which may have the answer to the above requirements.

1.2 Objective

This thesis aims to address two conceptual challenges. The first is the careful analysis of the design-space of fault-tolerance solutions for dataflow computations. This will involve relating previous research on rollback recovery to the requirements of users of dataflow systems.

The second is the design of a novel approach to the recovery of stateful operators in dataflow computations. This approach will be based on the promising concept of causal logging, as a way to address the issue of nondeterminism. The solution should provide the following improvements on current solutions:

- Allow for fine-grained recovery of single operators, rather than of the entire computation graph.
- Allow for faster recovery than current approaches, while still guaranteeing consistency.
- Allow for disconnected branches of the computation to make progress and output independently.
- Achieve minimal overhead during failure-free operation.

- Provide transactionless exactly-once delivery.

To ensure these requirements are met, a prototype will be implemented on a state-of-the-art streaming system. The approach will then be evaluated on several metrics and compared to existing solutions.

1.3 Expected Contributions

The main contributions of this thesis are as follows:

- An analysis of the state-of-the-art and the design-space of the fault-tolerance of dataflow systems.
- The design of a novel approach to fault-tolerance in distributed streaming systems, ensuring fast fine-grained recovery and exactly-once processing.
- The evaluation of the approach and experimental comparison with modern approaches to fault-tolerance in distributed dataflow systems.
- The contribution of the produced artifacts to the Open-Source community.

1.4 Document Organization

The remainder of this document is organized thusly:

- Chapter 2 begins with a brief introduction to the message-passing system abstraction and a review of common distributed systems topics such as virtual time. A lot of effort is then put into summarizing the literature on rollback-recovery, the main topic of this thesis. Finally, the state-of-the-art in dataflow systems is presented, along with some other work on selected topics.
- Chapter 3 starts by describing the desired algorithm and shows how causal logging naturally fits as an approach to fault-tolerance in dataflow systems. Following, the requirements of the system are specified and ranked on importance leading to the scheduling of the work to come.

Chapter 2

Related Work

In this chapter, work related to the proposed thesis is reviewed. It begins by reviewing some preliminaries related to distributed and concurrent systems in Section 2.1. These preliminaries then allow for the discussion of the class of rollback recovery algorithms in Section 2.2, to which causal logging belongs. Finally, in Section 2.3 dataflow systems are presented, and existing systems are examined regarding their fault-tolerance and use of rollback recovery algorithms. High availability in stream processing is also briefly addressed. The chapter is concluded with some lessons learned.

2.1 Message Passing Systems

An asynchronous message passing system is a common abstraction in distributed systems algorithms[23, 29, 43]. In such a system, N processes communicate exclusively through message passing to achieve some computation. They communicate in an *asynchronous* manner, meaning no bounds can be placed on time taken to process or receive a message. Communication channels are assumed to be *reliable* and *FIFO*, as such we do not concern ourselves with the possibility of message loss or duplication, nor with the possibility of messages being delivered in the wrong order. Messages are first *received*, but are processed in the order they are *delivered*. Processes fail independently, in a *fail-stop* fashion, meaning that they fail silently. They may later recover and rejoin the distributed computation. Each process has access to a form of *stable storage*, whose contents survive failures. The form of stable storage need not be defined, but it may range from a magnetic disk drive to a highly distributed filesystem. Of course, *volatile* memory is also present, and access to it is assumed to be faster than that of stable storage.

Useful systems obligatorily communicate with the outside world, for example storing records in a database, sending a request to the sprinkler control system to activate, or writing to a terminal. We can model the entire outside world as a process, called the *Outside world process*, to which messages can also be sent.

2.1.1 Events

The transmission and reception of messages are events in a message passing system. A process always transmits a message in a deterministic fashion, it always knows it will send a message. However receiving a message is a nondeterministic event[6, 11], because a process does not know that it will happen. If two processes send messages to a third process, these may arrive in any order, which may affect the result of the computation. This is examplified in Figure 2.1, where P_0 and P_2 both message P_1, however, these messages may take arbitrarily long to arrive, and thus may arrive in any order.

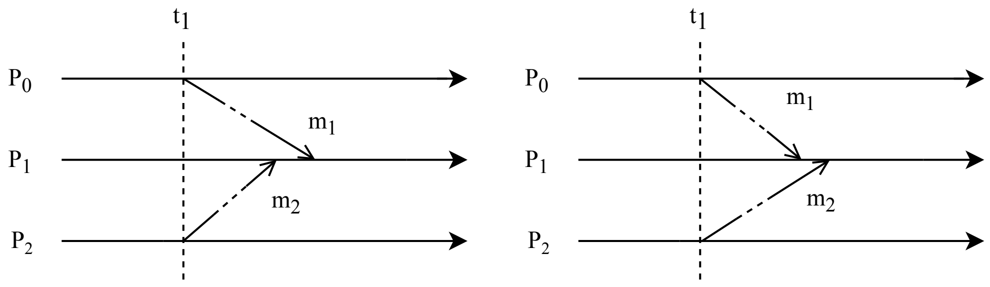

Figure 2.1: Nondeterministic order of arrival.

2.1.2 Virtual time and Digital Clocks

In order to build distributed protocols it is often very important to be able to order events. Synchronizing physical clocks in a distributed system is hard[43], so physical timestamps cannot always be used. Because of this, a notion of logical time needed to be developed. They were developed from the need to claim that an event preceded another.

Within a single sequential process, it is easy to see if one event happened before another. However, with a logical notion of time, it would sometimes be impossible to claim that one event on one process happened before another on another process, or the other way around. Because of this, the "happened-before" relationship is a partial ordering[43], denoted $\rightarrow$, defined as the smallest relation satisfying:

1. If a and b are two events within a single process, if a comes before b, then $a \rightarrow b$.

2. If a is the event representing sending a message, and b is the event representing receiving a message, then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two events a and b are concurrent if $a \nrightarrow b$ and $b \nrightarrow a$.

Lamport clocks[43] are a method for ordering events in a distributed system, such that if an event a causally precedes an event b, then the timestamp $C(a)$ is smaller then the timestamp $C(b)$. Formally, $a \rightarrow b \Rightarrow C(a) < C(b)$. To implement them, processes follow the following protocol:

- Whenever a process i executes an internal event, its clock ticks: $C_i := C_i + 1$.
- Whenever a process i sends a message to another process, it piggybacks the timestamp of the send event, which is its current clock value.
- Whenever a process i receives a message from process j, it combines the timestamp received t_j with its own clock, by taking the maximum of the two: $C_i := max(C_i, C_j)$

However, Lamport clocks do not provide the reverse guarantee, $C(a) < C(b) \Rightarrow a \rightarrow b$, because even though one timestamp may be larger than another, the two events could be concurrent.

Vector clocks[45] are a generalization of Lamport clocks, which offers this guarantee. Each process i maintains an n-long vector clock C_i, such that $C_i[j]$ indicates the last event from j that causally affects i's current state. To achieve this, processes follow the following protocol:

- Whenever a process i executes an internal event, its clock ticks: $C_i[i] := C_i[i] + 1$
- Whenever process i sends a message to another process, it piggybacks its vector clock.
- Whenever process i receives a message from process j, it combines the vector received t_j with its own clock, by taking the component-wise maximum: $\forall k \in 1..n . C_i[k] := max(C_i[k], t_j[k])$

By having each process track the clocks of other processes in its own timestamps, we can now order events which with Lamport clocks would have been concurrent.

2.1.3 Consistency

A global state in a system such as described is the collection of the states of the individual processes and their communication channels. This global state is consistent if when the state of one process reflects the reception of a message, the state of the sender also reflect sending that message[23, 29]. Essentially, in the global state, a received message must have been sent, however a sent message may be *in-flight*, meaning it has been sent and received, but not yet delivered. The transmission and delivery of messages are events.

A common way to visualize this is with cuts through system state as shown in Figure 2.2.

In this Figure, state S_1 is consistent with S_2 even though the sent message has not been received. However state S_0 is not consistent with state S_1 because a message which was received has not yet been sent. Thus as a whole, this cut is not consistent.

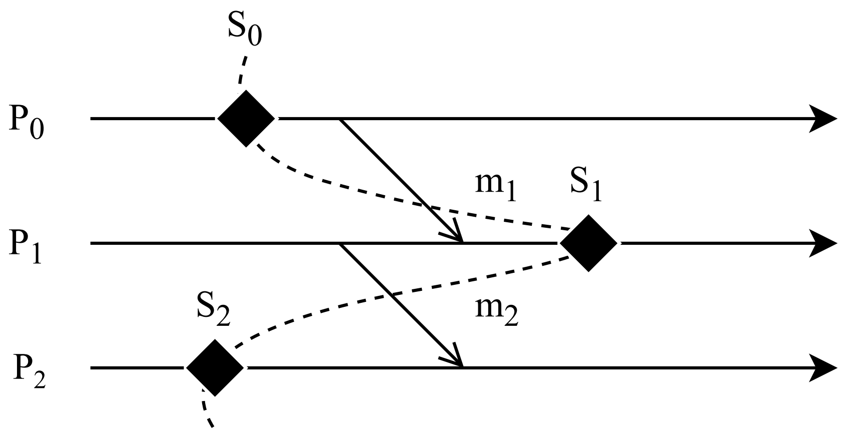

Figure 2.2: Inconsistent cut through a system

2.2 Rollback Recovery

Rollback recovery is a collection of methods to recover a consistent global state of a message-passing system after the failure of a set of processes. Recovery of consistency after a failure is achieved by executing a recovery algorithm, which will often roll back the state of some processes, such that all processes are consistent. However, we cannot roll back the state of the outside world, as the outside world is a process that does not participate in recovery. As an example, imagine a fire detection application. Once the sprinklers have activated, you cannot roll back the fact that this happened. This is known as the *output commit problem*. Because of the output commit problem, when a process sends a message to the outside world, it must make sure that the state from which that is done is recoverable in the case of failures. Rollback recovery algorithms store information on stable storage, before output commit, so as to ensure that the state is recoverable.

As a note, most of this section is heavily based on the excellent survey by Elnozahy et al.[29]. This section begins by introducing checkpointing-based approaches, then goes in detail on log-based approaches, with special interest in causal logging approaches, as it is a central component of this work.

2.2.1 Checkpointing-based approaches

Checkpointing approaches[8, 23, 29] have the participating processes occasionally take *checkpoints* (or snapshots) of their state to stable storage. A consistent set of checkpoints makes a global consistent checkpoint. When a process fails, normal execution is paused and a recovery algorithm is executed. When the failed process recovers, its previous state is lost, and it can only load one of its available snapshots from stable storage. However, to resume normal execution the participating processes must be in a consistent state. The job of the recovery algorithm is to calculate the latest consistent set of checkpoints available, called a *recovery line*. To conclude the recovery algorithm, each process will load their respective checkpoint of the recovery line, and execution resumes.

Checkpoints divide the execution of a process into intervals. The j'th checkpoint of process i is denoted $C_{i,j}$. Between two consecutive checkpoints $j-1$ and j of process i, an interval $I_{i,j}$ is defined.

2.2.1.1 Coordinated or uncoordinated

Distributed checkpointing algorithms can be either coordinated[23] or uncoordinated[8]. Coordinated protocols immediately create a new recovery line by ensuring that processes take checkpoints at the same logical point in time. Uncoordinated protocols allow processes to take checkpoints at any time, and thus must track dependencies between checkpoints in order to compute the recovery line. Uncoordinated checkpointing allows each process to choose the time to checkpoint which is most convenient to it, such as when its state is smallest. Generally, uncoordinated checkpointing protocols incur overhead from tracking causal information during normal operation, while coordinated checkpointing incur coordination overhead, though the coordination overhead has been shown to be negligible[29]. Thus, coordinated checkpointing algorithms have a simple recovery algorithm, while uncoordinated algorithms must calculate the recovery line.

Uncoordinated protocols must be designed carefully to avoid the *domino effect*, which may happen if no recent recovery line can be computed, due to the fact that there is always a message reception in a processes checkpoint which has not yet been sent in another processes checkpoint. This concept is illustrated in Figure 2.3, where due to a failure in P_1 the recovery line begins with the three most recent checkpoints $C_{0,3}, C_{1,2}, C_{2,2}$. because m_4 has been received in $C_{0,3}$, but not sent in $C_{1,2}$, we must rollback the 0'th process further, to checkpoint $C_{0,2}$. This rollback leads to the fact that m_3 is now received in $C_{2,2}$, but not yet sent in $C_{0,2}$, so we must rollback the second process to checkpoint $C_{2,1}$. This process may continue indefinitely, until we reach the start state of the system.

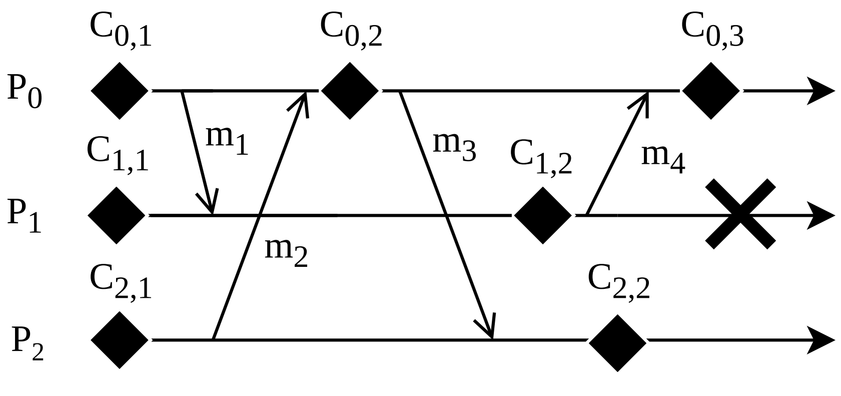

Figure 2.3: Example where uncoordinated leads to domino-effect

Several variations on uncoordinated checkpointing have been developed, which aim to avoid the domino effect, such as communication induced checkpointing and model-based communication induced checkpointing[8]. These protocols piggyback even more information on their messages, which is used to decide when to force checkpoints to avoid heavy rollback.

2.2.1.2 Blocking or non-blocking

In coordinated checkpointing, one must ensure that the individual checkpoints happen in a consistent point in time. To be explicit, for a message m sent from i to j, if $C_{j,k}$ reflects the event $receive(m)$, then $C_{i,k}$ must reflect $send(m)$.

A simple checkpointing protocol[51] may behave like a two phase commit protocol[16], where an initiator first takes a checkpoint and then sends a checkpoint request to all other processes. When processes receive this request, they perform their own checkpoints (including the state of their channels), and acknowledge the initiator of their success or failure in performing the checkpoint. The initiator can then decide whether the global checkpoint is valid or should be discarded. When processes are informed of the decision they may resume processing. This is a *blocking* algorithm, because during the execution of the checkpoint, processes stop their computation.

The alternative are *non-blocking* algorithms, such as Chandy-Lamport's distributed snapshot algorithm[23]. This algorithm builds on the assumption that FIFO channels will order the checkpoint markers with the application messages. The initiator begins by taking a checkpoint of its state and then sending a marker along each of its channels. Upon receiving the first marker on a given channel, a process checkpoints its state and marks the state of that channel as empty. The process also sends a marker on each of its outgoing channels. It then records incoming messages on other input channels, up to the point where the markers arrive. The result is a consistent global checkpoint, including the state of the individual processes and their channels.

2.2.1.3 Synchronous or asynchronous

Blocking and non-blocking checkpointing is distinguished from *synchronous checkpointing* and *asynchronous checkpointing*. Asynchronous checkpointing happens when the act of flushing the state to stable storage is not on the critical path of the system, such as when it is done by a separate thread. This can be achieved by taking an in-memory copy of the state, and spawning a thread to asynchronously flush it the copy, while simultaneously working with the original data.

2.2.1.4 Incremental checkpointing

In order to further reduce the overhead of checkpointing the state of an application, certain systems choose to provide *incremental checkpointing*[19, 29]. With this feature the changes performed to state since the last checkpoint are collected and act as the next checkpoint. An incremental checkpoint is the difference between the current state and the previous checkpoint. This works best in an application with very large state, but comparatively low amounts of modifications to it. A compaction algorithm must be run occasionally to combine several incremental checkpoints into a full checkpoint. This reduces memory consumption and eases searches.

2.2.2 Log-based approaches

A process's lifetime can be seen as a sequence of intervals separated by nondeterministic events. Inside a state interval, a process evolves independently and deterministically. Sending a message from one process to another then creates a new state interval for the receiving process. The senders state interval is then causally linked with the receivers new state interval. Note that, for any given execution, if all nondeterministic events of a process were replayed in the same order and at the same points in the execution, then the execution inside the intervals would also be the same, as it is deterministic.

Log-based rollback approaches replay the nondeterministic events generated pre-failure in order to recover the failed processes to a consistent state with the remaining system. In some cases this may completely avoid rolling back the state of other processes. Log-based approaches can be combined with checkpointing approaches to allow limiting the size of the produced log, as one may delete the log entries older than the checkpoint.

These approaches rely on the *Piecewise Deterministic Assumption* (PWD assumption)[29]. This assumption states that all nondeterministic events can be identified and their determinants logged. To reproduce a nondeterministic event e, one must store the event and its *determinant*, represented as #e, which is a piece of data that removes the nondeterminism of an event. As an example, suppose the event is generating a random number, the determinant of that event is the number generated. If the nondeterministic event is receiving a message, then the determinant is the order of delivery. In other work from the same authors[7], the PWD assumption is stated differently as *"The only source of nondeterminism is the order of delivery of messages"*. This other definition is fundamentally different as in most realistic use-cases other forms of nondeterminism are needed. In this thesis, we refer to this other assumption as *Ordered-Delivery Deterministic Assumption* or ODD assumption. This name is chosen because if the order of delivery was fixed, the system would be deterministic.

Thus, if a process re-executes the same nondeterministic events, in the same order, it will evolve to the same state as before failure. If all determinants are available, other processes need not be rolled back, as the failed process will reach its pre-failure state.

2.2.2.1 Orphan Processes

An orphan process is defined as a process whose state depends on a nondeterministic event that cannot be reproduced during recovery[29]. If a nondeterministic event cannot be reproduced, then the state of all processes must be rolled back to before that event, in order to provide consistency.

The *always-no-orphans* property[6], is used to reason about the effects of a nondeterministic event e:

$$\forall e : \Box(\neg Stable(e) \implies Depend(e) \subseteq Log(e)) \tag{2.1}$$

Where Depend(e) is the set of processes whose state was affected by e, meaning any process that executed e or has since received a message from the process that executed it. Log(e) is the set of processes that have logged e's determinant in volatile memory and Stable(e) is a predicate which becomes true when e's determinant is logged in stable storage. Finally, the operator $\Box$ is the temporal always operator. It is thus apparent that this is a safety property of the system.

This condition states that to ensure no orphaned processes at all points in the execution of our distributed computation, for every nondeterministic event e that a process executes, if its determinant is not yet in stable storage, then all processes whose state depends on event e must have logged its determinant in volatile memory.

2.2.2.2 Pessimistic logging

Pessimistic logging protocols[29], such as in the write-ahead logs of database literature [16], implement a stronger property than the always-no-orphans property[6]:

$$\forall e : \Box(\neg Stable(e) \implies |Depend(e)| \leq 1) \tag{2.2}$$

Essentially, for a process to be able to send a message to another process, it must first ensure that all determinants are stable. The cost of accessing stable storage is often prohibitively high, however these protocols come with several attractive advantages. First, any process may send messages to the outside world without coordination. Second, any state in which the system is observed is recoverable. Rollback of other processes is unnecessary and recovery is simple. However, in general, failure-free execution has very high overhead.

2.2.2.3 Optimistic logging

Optimistic logging protocols[29, 50] have the lowest overhead during failure-free operation. They achieve this by allowing for the temporary creation of orphans, but guaranteeing that by the time recovery is finished, no orphans will exist. Thus they do not ensure the always-no-orphans property. The property they provide is shown in 2.3[6], where $\mathcal{F}$ is the set of processes assumed to fail concurrently during execution and $\diamond$ is the temporal eventually operator:

$$\forall e : \Box(\neg Stable(e) \implies (Log(e) \subseteq \mathcal{F} \implies \diamond(Depend(e) \subseteq \mathcal{F}))) \tag{2.3}$$

This property states that if a determinant is not stable, then if all the processes which have logged it fail, then eventually only the processes which have failed will depend on it. This essentially means that after a failure, eventually there will be no orphans, because processes will roll-back until their state does not depend on a lost determinant. These protocols are implemented by occasionally logging their determinants to stable storage, often asynchronously. During failure-free operation they must track causal dependencies between process's state intervals, such that during recovery, a consistent state may be achieved. Though they have low failure-free overhead, optimistic protocols come at the cost of slow output commit, which requires a lot of coordination and complex recovery.

2.2.2.4 Causal logging

Causal logging[11, 28] attempts to get the best of both worlds, with low overhead, the ability for each process to independently commit output to the outside world and ensuring the always-no-orphans property, while also removing access to stable storage from the critical path, except when committing to the outside world.

While pessimistic logging ensures no orphans by ensuring the antecedent of property 2.1 is true, causal logging focuses on ensuring the consequent. That is, causal logging ensures that all processes that depend on an event have logged its determinant. If a set of processes $\mathcal{F}$ fails, then for all events e either $Depend(e) \subseteq Log(e) \subseteq \mathcal{F}$, in which case there is no orphans, or $Depend(e) \subseteq Log(e) \not\subseteq \mathcal{F}$ in which case at least one surviving process has the determinant of e, and can share it with the recovering processes. In order for a process to message the outside world, it must ensure that the determinants it depends on are stable, however this can be done with no coordination.

These protocols can be made optimal by ensuring that no unnecessary determinants are sent to processes that do not depend on them. This is done by strengthening the always-no-orphans property, as shown in property 2.4:

$$\forall e : \square(\neg Stable(e) \implies ((Depend(e) \subseteq Log(e) \wedge \diamond(Depend(e) = Log(e))))) \tag{2.4}$$

This can be interpreted to mean, while e is not stable, all dependent processes must have logged it, and eventually the ones who have logged it will be exactly those who depend on it and no more.

However, processes only depend on events of other processes if they receive application messages from those processes, because those events happened-before the delivery of the message. It should thus be evident that there is no need to send extra messages containing determinants, since the determinants a process needs can be piggybacked on the message that makes it causally dependent on those determinants.

Finally, if the number of possible concurrent failures is bound to not be greater than f, it is possible to implement stable storage while avoiding disk access by logging to f + 1 processes. This property is shown in 2.5

$$\forall e : \square((|Log(e)| \leq f) \implies ((Depend(e) \subseteq Log(e) \wedge \diamond(Depend(e) = Log(e))))) \tag{2.5}$$

Of course, in this case, one process may avoid sending its determinants to processes which have not logged them if enough processes have already logged them, such that they are now considered stable.

In the following, a general protocol for causal logging is described from the perspective of process p. Process p maintains a determinant log L, which is a mapping from processes to the determinants of the events on which the state of p depends. Processes maintain additional data-structures where they record which determinants are stable and have been received by which processes. Additionally, processes piggyback determinants on application messages, sending the determinants they believe the receiving process to not yet possess.

1. When a process receives a message m, before delivery, for all determinants piggybacked on m, which originate in process q, p appends to L[q] those determinants.
2. When a process delivers m, it records in L[p] the delivery sequence number.
3. During the processing of m, more determinants may be generated (random numbers, timestamps), which are appended to L[p].
4. Whenever p wishes to send a message to a process q, it will consult its log and data-structures, to compute the set of determinants that q should receive. These are piggybacked on the message sent.

Two special cases of optimal causal message logging protocols exist. *Family-Based Logging*[9] is the special case where $f = 1$. The name comes from the fact that since at most one process can fail at any time, it is sufficient to log the determinants in the direct *children* of a process, which are the processes to which a process sends messages. The processes from which a process receives messages are its *parents*. These have the responsibility of performing *sender-based message logging*[40], such that messages may be replayed during recovery.

The other special case is when $f = N$, which is the protocol that Manetho[27, 28] implements. Manetho, like most other work in the area of causal logging, works on the ODD assumption, that is, the only source of nondeterminism is order. Since all processes may fail, determinants must be completely shared, which can be done by tracking message deliveries only. Manetho does this with the antecedence graph, an efficient datastructure that tracks the state intervals of processes. Manetho also uses checkpointing to be able to truncate the antecedence graph. Checkpoints are uncoordinated and include the in-flight messages, the antecedence graph and the state of the application. In [28], it is shown that causal logging has low failure free overhead experimentally.

Intuitively, this approach recovers failed operator o, by having it send the non-stable determinants it has generated piggybacked on every message it sends. This means that other processes, if causally affected by o, will know how to guide its recovery. They can share with it, for example, the order in which o previously received messages, such that it can reach the same state as before.

Causal logging protocols track the causal dependencies between state intervals of processes. Each process logs the determinant of every non-stable event that causally affects it. Thus, each process maintains a determinant log, which acts as an insurance against failures of other processes. In Figure 2.4, an example is shown, combined with uncoordinated checkpointing. When a process takes a checkpoint, determinants it had previously logged become stable. This is why P0 does not have #m2 in its log, since P2 has logged it in stable storage. P2 took a checkpoint so it could send a message to the Outside-World Process, this was done without coordination. As an example, if P2 failed, P2' would recover from its checkpoint. P2' than requests and receives the replay of messages sent since its last checkpoint. At this point, P1 can guide its recovery, because it knows that m_5 was delivered before m_6.

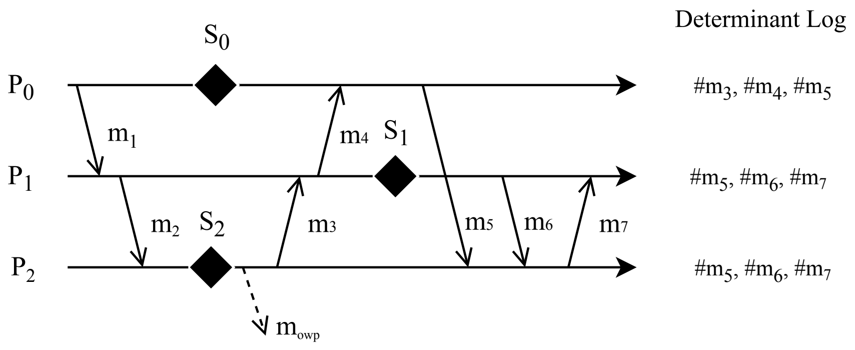

Figure 2.4: Causal logging example

Other ways in which causal logging protocols may vary are the way in which they track the number of processes that may have logged a given determinant. This is addressed in the following section.

2.2.2.5 Trade-offs in membership tracking

Whenever a distributed system follows the ODD assumption, which is that the only source of nondeterminism is message delivery order, the determinant of a message m, denoted #m, can be captured by the tuple $< m.src, m.ssn, m.dest, m.dsn >$. Src and dest are respectively the sender and receiver of the message, while ssn and rsn are the send sequence number and the delivery sequence number. This is of course assuming that the sender remembers the messages sent. These sequence numbers then allow a process to request replay from another process, starting at a given send sequence number. The deliver sequence number says the order of delivery, required for delivering in the same order.

In order to avoid redundantly sharing unnecessary determinants, processes may share some metadata regarding who knows about what determinants. This would allow processes to eventually stop sending determinants to processes that have already received them. But often, the amount of data to be sent is large, and to reduce the amount of determinants sent, more metadata must be sent. In [10], the authors explore this exact trade-off, presenting six causal logging protocols which share different amounts of data, under the ODD assumption:

- $\prod_{det}$: A process p tracks who has received which determinants from itself only. If they have not yet received them, then they are piggybacked on the next message. No aditional metadata is sent.

- $\prod_{|log|}$: A process p tracks the number of processes that have logged which determinants. When process p receives a determinant of a message m sent by process $m.src$ to process $m.dest$, and p is neither of those processes, then p knows that $|log(m)| \geq 3$. A process which receives this metadata from p then knows that $|log(m)| \geq 4$. When $|log(m)| > f$, processes stop sending the determinant.

- $\prod_{log}$: A process p tracks the identifiers of the processes that have received a given determinant. This way, a process will never send a determinant to a process it knows has already received it once. When receiving a determinant, the union of the two sets of identifiers is taken. When $|log(m)| > f$, processes stop sending the determinant.

Additionally, versions of the above protocols are specified, which additionally inform other processes of changes in the stability of storage of a determinant:

- $\prod_{det}^{+}$: Additionally, informs other processes of which determinants have become stable from its perspective.

- $\prod_{|log|}^{+}$: Additionally, informs other processes it had already told about the determinant of m of changes in $|log(m)|$.

- $\prod_{log}^{+}$: Additionally, informs other processes it had already told about the determinant of m of changes in $log(m)$.

While $\prod_{Det}$ would piggyback much less information on each message, it runs the risk of sending determinants to processes that have already received them. Additionally, for $f > 3$ this protocol is not able to recognize that a determinant is stable, it must be explicitly informed. This is because, since no metadata is sent, a process receiving #m, can only assume that the holders of #m are itself, and $m.src$. On the other hand, $\prod_{Log}^{+}$ will rarely send redundant determinants, and will know a determinant is stable as soon as possible, but will piggyback a lot of metadata in order to achieve this. In general however, $\prod_{Det}$ is a good choice for $f < 3$, since only determinants need to be sent, and processes are able to recognize that they are stable.

Another good use-case is shown for when the channel graph, the graph of communication channels, is acyclic and shortcut-free. Then $\prod_{Det}$ is as efficient as $\prod_{Log}^{+}$, when $f = N$. If the channel graph is additionally a tree (meaning it is acyclic, shortcut-free and each node has only one parent, except for the root) then this is true for $f \leq n$.

In order to efficiently propagate the information of the protocols, a dependency vector may be used, first introduced in [50]. First note that in a piecewise deterministic system, |Depend(e)| may be used to estimate |Log(e)|, since Depend(e) ⊆ Log(e). Additionally, remember that processes are assumed to be deterministic, except in the order of delivery of messages. This means that each state interval is started by a deliver event. In [7], the approach is explained in detail, we urge the interested reader to refer to it, as it is heavily condensed here. By having each process i maintain a vector clock, called a dependency vector DV_i, which increments only on deliver events, it is possible to know which events causally precede a certain event e. For two messages m and m', delivered to processes p and q respectively, the following holds, because the DV are vector clocks:

$$deliver_p(m) \rightarrow deliver_q(m') \equiv DV_p(deliver_p(m))[p] \leq DV_q(deliver_q(m'))[p] \qquad (2.6)$$

This means that $DV_p(deliver_p(m))[q]$ is the index of the latest state interval of q which affects p. To know whether a process depends on a determinant, one can use the following implication: $DV_q[m.dest] \geq m.dsn \Rightarrow q \in Depend(m)$.

To track dependencies between processes, each process maintains a $N * N$ matrix called DMat. Process p maintains in row p its dependency vector, and in all other rows, its estimate of the other processes dependency vectors. To keep it up to date, whenever process p receives a message m (with attached metadata) from q, it executes an update rule, different for each protocol.

As an example, to implement $\prod_{det}$, the metadata sent, which efficiently encodes all determinants, is simply an N long vector named PBC(m), where PBC(m)[p] is the maximum $m.dsn$ for all determinants #m piggybacked in the message, where $m.dest = p$. To update it, p first increments $DMat[p,p]$, then sets DMat's p'th row to the component-wise maximum of itself and v, and does the same for the q'th row and v. Finally, it updates the diagonal of the matrix to the maximum of itself and the corresponding entry in v.

To implement $\prod_{log}^{+}$, the full DMat matrix is piggybacked on each message sent from q to p. When p receives the message m it first increases $DMat[p,p]$, then takes the component-wise of the two matrixes $DMat_p$ and $DMat_q$.

A process p may estimate the size of Log(m) by counting the number of processes q such that $DMat[q, m.dest] \geq m.dsn$. More intuitively, by looking at the m.dest column and counting the entries equal to or above the original receive sequence number of the message, which is also the state interval initiated. When this value is above f, process p may consider m to be stable.

2.3 Dataflow Systems

Dataflow programs structure their processing as flows of data between a directed acyclic graph (DAG) of operators[5, 14, 35]. Each of these operators applies a transformation to a collection of data items. Structuring programs this way allows for easier use of several forms of parallelism, as will be discussed later.

In this work we are particularly interested in distributed dataflow systems, which execute dataflow programs, due to their scalability and fault-tolerance. Two other important components that led to the popularity of these systems is that they generally offer transparent recovery mechanisms, alleviating users from implementing complex fault-tolerance logic, and also automatically schedule tasks with data locality awareness[38, 60]. For example, the success of MapReduce came not from being the first system to distribute computation, but from providing a system in which Map and Reduce functions could be written as if they would execute locally.

From dataflow systems two categories are highlighted: *batch* processing systems, which operate on *bounded* collections of data, that is data which is finite and fully present such as a file or database table. *Stream* processing systems, on the other hand, operate on *unbounded* collections of data, called *streams*, which are continuously arriving and must be processed online.

Dataflow computations, especially streaming ones, may be viewed as message-passing systems, which allows us to pull from the rich literature on rollback recovery. To keep discussion clear, nodes and messages will refer to message passing systems, while operators and records will refer to dataflow computations.

Importantly, dataflow systems must somehow receive data from an external system, and put results into an external system, both of which can be represented by the outside world process. The place where data is received from is called a *data source*, which should not be confused with a *source operator*, the operator from which records appear in the graph. Similarly, the place where data is put after processing is a *data sink*, which also has a corresponding *sink operator*. These may often be a database, message queue or even highly distributed filesystem.

Consistency guarantees are an important feature of dataflow systems, as they affect latency, throughput and are generally desired by users. These guarantees refer to how many times a record is counted in some way, the possibilities being at-most-once, at-least-once and exactly-once. Two types of semantics are important when discussing dataflow systems:

- **Processing semantics**: The guarantees offered by the system as to how many times the input records will affect the internal state of the system [19]. At-least-once processing semantics means that each input record will be processed (and thus affect internal state) at least once by each vertex in the computation graph.

- **Delivery semantics**: Often also called end-to-end processing semantics, these are the guarantees given across the internal system and its connections with the outside world process. Exactly-once delivery would mean that each record is fetched once from the outside world data source, and sent once to the outside world data sink.

Delivery semantics can only ever be as strong as the processing semantics.

Before diving into the central topic of this work, streaming systems, batch systems are first reviewed.

2.3.1 Batch Computational Model

Batch dataflow systems specify computations on a bounded static dataset. These computations are specified in terms of a DAG of operations to be performed on the dataset. The operators are specified as functions which operate on a single input record, while records are partitioned equally among participating vertexes of any given stage. There are some implicit input operators which are in charge of injecting data into the DAG. Connections between operators express data dependencies. Each operator in the DAG may only begin processing once all upstream operators have finished processing. Since these systems are generally meant to process extremely large datasets, these datasets are generally already partitioned and replicated in a distributed file system. Batch Systems generally take advantage of this by scheduling computation on the nodes which have the data that needs to be processed. One final common property of these systems is the use of *lineage* to recover from failures.

2.3.1.1 Lineage Recovery

Another form of recovery mechanism, very common in batch dataflow systems, which is not a rollback-recovery mechanism is lineage-based recovery. Lineage based approaches[55, 59] differ from rollback-recovery approaches in that they only apply to dataflow systems with large coarse-grained computations or tasks which are applied to partitions of data, and not to general message passing systems. The output of each coarse grained transformation on the input is stored in distributed stable storage. Lineage reconstruction is used to reconstitute the state of a distributed computation by storing the coarse grained operations performed on the previous state or dataset. This creates a lineage graph, which may be partitioned. To reconstitute partitions after a failure, one may simply apply the same transformations to the input set. Several systems implement this kind of recovery for its low runtime overhead and simple implementation, such as MapReduce[25], Spark[60] and Ray[46], all described below.

2.3.1.2 Existing Batch Systems

MapReduce[25] was the first marjorly popular instance of this concept. It allowed a computation to be specified in terms of a Map and a Reduce operation, with an implicit shuffle of data in between. Mappers acted as input vertexes, reading data from Google

File System[31], while reducers acted as outputs, writing to it. Intermediate results would be written to disk. Depending on the task, the reduces may be dependent on all maps, meaning a single “straggler” map may slow down the entire computation. Both stragglers and failed tasks are dealt with by rescheduling. Later Hadoop, an open-source implementation of MapReduce, became massively popular. Hadoop replaced GFS with the Hadoop Distributed File System[48] (HDFS), a scalable distributed file system. MapReduce as framework was inflexible, allowing only one stage of Map and Reduce operations. As requirements became more diverse, users began chaining several jobs to achieve more complex behaviour. Eventually this led to the appearance of frameworks such as Hive[52] and Pig[47], which allowed queries to be expressed in higher level, more expressive languages. Nonetheless, intermediate stages of these chained computations were inefficient, as they had to fully complete before starting the next, while also writing every intermediate result to HDFS.

Dryad[38], a research system from Microsoft, provided a much more expressive computational model than Hadoop. Any computational DAG can be expressed, with virtual input and output operators. Any function may be executed by a vertex, though to provide fault-tolerance functions should be deterministic. Vertexes are rerun when failures happen. Importantly, if the inputs to a vertex disappear, then recursively the upstream vertexes are rerun, in which case vertex functions must be kept deterministic to avoid rerunning siblings which may have already completed. Again, though not explicit, this is essentially an implementation of lineage recovery. Similarly to MapReduce, it takes advantage of data locality and a distributed file system. The edges between vertexes represent channels, whose type must also be specified, such as TCP, UNIX pipes or shared memory. This flexibility is also a source of complexity, and perhaps due to that DryadLINQ[30] was later developed, which allowed queries in a SQL-like language to be compiled into Dryad flows. Also like MapReduce, a vertex must wait for all its inputs to be ready before beginning computation. This means that it takes as long as the slowest input to start. Dryad automatically detects slow vertexes and schedules new copies to compensate. Due to its closed-source nature, Dryad never grew much in popularity.

The massively popular **Spark**[60] appeared a few years later, borrowing many ideas from Dryad, however building them on the open-source HDFS. One important contribution that Spark made was introducing the resilient distributed dataset. This abstraction allows users to write dataflow programs as simple sequential programs, while also providing fault-tolerance through automatic lineage tracking and recovery[59]. Another important factor for the popularity of this system was the ability to perform in-memory data-mining[60].

2.3.2 Streaming Computational Model

Streaming dataflow systems also express their computations as a DAG of operators, but since data is unbounded, it must be continuously processed. Thus in a streaming computation, all operators in the DAG are concurrently processing. This means that while batch processing exploits task and data parallelism, streaming additionally exploits pipeline parallelism. Operators process records one by one, receiving them from input streams and outputting them into an output stream. The operators used are like those of relational algebra though recently user-defined functions have become popular.

To discuss the existing systems, we use and extend the taxonomy used in [14]:

- The job is described as a DAG of operators, which is translated into a *logical plan*. This logical plan is later translated into a *physical plan*, by specifying the parallelism of each operator, or allowing the system to choose one.

- Regarding selectivity, operators may be *selective*, *one-to-one* or *prolific*, where they respectively produce less, equal or more records than they consume.

- The operators may be *stateless*, when they do not maintain state between executions, or *stateful* if they do.

- The processing model may be *by-record*, where an operator is applied to a record at a time, or *micro-batch*, when a small batch is passed between operators.

- Operators may be *deterministic*, when they apply a deterministic function over their inputs, or *nondeterministic*.

- The source operator may be *pull-based* or *push-based*, depending on whether it pulls data from the outside world or if the outside world pushes data to it.

2.3.2.1 Streaming Concepts

Streaming brought with it a new set of operations and concepts which are still being refined to this day. One of these concepts is windowing. Windows are operations which execute on a finite set of records of a stream. These may apply any function over this set though commonly they are used for aggregates. Additionally, the set of records is also contiguous based on some notion of time. Windows have two properties, size and slide. Size defines how large the window is in units of the notion of time and slide defines by how much the window should advance in that notion, after each triggering.

Three notions of time are common:

- *Processing-time*: the physical timestamp at which the record is being processed in this execution.

- *Ingestion-time*: the time at which the record was ingested by the system. Since records may be created from processing other records, often their timestamps are combined either by taking the maximum or the minimum.

- *Event-time*: the time at which the record was produced at the event generator.

Event-time is very popular but raises the issue of out-of-order processing and lateness. For example, a user of a mobile game plays while disconnected. When he reconnects to the network, old events are sent to the streaming engine, late. To deal with this *watermarks*[3] have been proposed, which allow event time to advance by injecting a punctuation which asserts the current time in event-time. To read more on windowing and watermarks please refer to the excellent paper [5].

A streaming *punctuation*[54] is an element embedded into a stream which contains a predicate. Its presence at a point in the stream states that no following records will match the predicate. This has several uses, such as for watermarks or checkpointing, as we will see later. For use in watermarks, the predicate simply states that the event-time of following records will be above some number. This allows windowing operators based on event-time to fire windows whose size exceeds that time, and thus have bounded state.

It is very important to highlight that certain streaming operations are inherently stateful. Operations like streaming joins, windows and aggregations must maintain state between executions. These are also some of the most useful operations to have in a streaming framework. Their stateful nature is what makes providing high-availability with strong processing semantics incredibly hard. On the other hand, processing time windows are inherently nondeterministic, as their execution depends on the physical clock. If the execution of a processing time window was repeated, even with the same order of delivery, differences in delivery time may lead to different results.

2.3.2.2 Existing Streaming Systems

Arguably, the first Data Stream Management System was **Aurora**[22], which needed no specialized fault-tolerance as it was a single-node system. Later, **Aurora*** extended this work to provide distributed stream processing. Its input operators recorded records which arrived at them in a stable fashion, which is important to provide at-least-once processing since data sources would push data to it. Operators are distributed across nodes in a sensible manner, meaning continuous chains of operators are assigned to the same node. To simplify, assume each node gets one operator. To provide fault-tolerance, Aurora* used *upstream backup*[24] which to support f failures of nodes, would backup in-flight records of a node o on k upstream nodes. These messages were maintained by the backups until nodes downstream from o have fully processed these messages. When a failure is detected, an upstream node from operator o takes over the execution of operator o. Load-sharing techniques can then be applied to offload work from the node which is now running o. Importantly, operators also track which past records they depend on using a sequence number. The earliest record that an operator depends on is also communicated to its backups, so they may truncate their backup log. Note that stateful operators may depend on all records ever processed. Thus, this very early system provides high-availability with exactly-once processing at the cost of high synchronous communication and memory overhead with the upstream servers, however this scheme only works for line DAGs, that is computations that may be expressed as a line of operators. This is because, as shown in Section 2.1.1, order of arrival is nondeterministic. It is also important to note that only deterministic operators are supported, given that input messages must be replayed in the case of failures.

Borealis[1], a successor to Aurora*, provided high-availability by having replicas of each operator running simultaneously. The ODD assumption was followed, meaning operators were deterministic and that the only source of nondeterminism is the order of delivery of records. In order to make deterministic the order of delivery, an operator, named SUnion, is inserted into the graph where necessary. This operator takes multiple input streams and produces one deterministically ordered output stream. However, to ensure consistency through all replicas of SUnion, records must be buffered, increasing the latency. In the case of a failure of an operator o, operators downstream from o will search for a substitute replica. If found, they will proceed normally from the point at which o's operation stopped. If not found, then the user has the choice of pausing processing or continuing with partial input. Partial input leads to tentative records, which may later be revised. Whatever the users choice, the state of the failed operator and other downstream operators must be reconciled after recovering from the failure. Two methods of state reconciliation are described in [15], checkpointing and undo. Checkpointing works as expected, the graph is reset to a checkpoint and input replayed until the operator is up-to-date. The undo method involves taking the tentative records that the operator has created and undoing their effects, followed by recalculating the records with full inputs. This approach only affects the paths starting at the failed operator, and as such is finer-grained. If one ignores tentative records, then Borealis can be said to provide exactly-once processing semantics, though at the cost of having several passive replicas and increased latency from the SUnion operators.

D-Streams[61] is a streaming model implemented in Spark Streaming, where input records are bucketed into RDDs, called micro-batches. Stateful operators are supported, but are turned into stateless functions that accept a previous state as input. The output of one of these operators is both the original output and the current state. Both of these are also represented as RDDs. As explained before, RDDs provide transparent lineage recovery, meaning that fault-tolerance is immediately provided by this system. To prevent recomputing an operator's state from the start, periodic asynchronous checkpoints of the state RDDs are taken, which may then be used to bound lineage recovery. Due to its micro-batching nature, D-Streams incurs some overhead from the batching process, and latency is always at least as large as the micro-batch period. This also translates to all operators, which block while processing an input RDD. Thus the records in the output

RDD are all outputted at once, and the next operator cannot start processing them earlier. In [41], Spark Streaming is shown to perform and scale well in general, though often performing worse than modern purpose built stream-processors such as Flink. Again, D-Streams provides exactly-once processing but requires operators to be deterministic.

MillWheel[4] is a general purpose stream processing system, which popularized many concepts known today such as watermarks and assuming out-of-order streams. It also offered exactly-once delivery semantics, though this was only achievable through the use of specialized databases and hardware. Because of the high-speed networks Google has at their disposal, it is viable to execute a transaction every time a record is processed, which atomically commits changes to operator state and output records. This atomicity allows for operators to be nondeterministic. Another interesting mechanism used in MillWheel is their method for identifying duplicate ingestion. Each record is attributed a unique identifier and a bloom filter[17] to provide fast membership lookups. This of course has the drawback of being probabilistic, though it can be shown to fail incredibly rarely. MillWheel's approach is not suitable for good performance on commodity hardware, as it requires specialized databases with highly synchronized clocks.

Storm[53] was another general purpose stream processor. While it did not support stateful operators, source operators still checkpointed their offset of the data source. At-least-once processing is achieved through a unique mechanism: each record, even those created by operators, receives a 64-bit identifier. A special operator is placed in the query graph, connected to all operators, called the "acker". It tracks the lineage of record generation using an XOR checksum: when a source operator first retrieves a record, it sends that identifier to the acker operator, who creates a map entry from that input record identifier to the checksum. The checksum is initialized as the bitwise XOR of the record identifiers generated at the source. Every time an operator processes a record, which may generate new records, it applies the bitwise XOR to the result record identifiers, and sends that, along with the original record identifier, to be XOR'ed with the checksum, after which the input record is acked and is XOR'ed with the checksum for the second time. When the checksum becomes 0, this implies that the original record has been fully processed. If it does not become zero, a timer will expire, considering the record failed, which will cause it to be retried.

Flink[21] combines stream and batch processing in one system, though in this work we will treat it as a stream processor alone. It has gained popularity due to its expressiveness, performance and configurable guarantees[41]. It is capable of exactly-once processing when combined with a replayable data source such as a Kafka, as explained in Section 2.3.4. Consistent checkpointing is achieved through the use of a modified Chandy-Lamport[19, 23] distributed snapshot algorithm. A coordinator injects punctuations into the streaming graph sources, which divide the records into *epochs*. When an operator receives this marker on a channel, that channel is blocked, while others keep processing. When received on all input channels it causes the operator to take a local snapshot. The marker is then sent downstream on all channels. Sources checkpoint their offset in the data sources. Figure 2.3.2.2 represents this checkpointing algorithm. Note that there is no need to record the state of the channels. Additionally, in Flink these snapshots of local state are done asynchronously, by performing an in-memory copy of the operator state, which is then asynchronously flushed to stable storage[20]. Finally, incremental checkpoints are also possible.

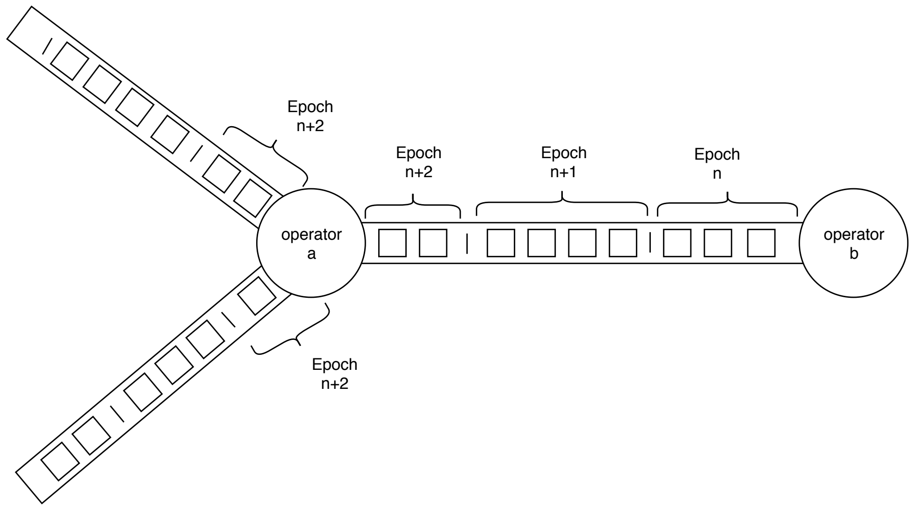

Figure 2.5: Checkpointing in streaming implemented using punctuation.

Streaming systems have been converging on this approach to fault-tolerance, as through the use of punctuation, one can achieve low latency and coordination. Flink was not the first to use this approach, though it did improve on earlier attempts, such as IBM Streams [39] which uses an unnecessary draining phase for in-flight records.

More recent systems have recognized the value of causal logging and have begun applying it. **Noria**[32] is a web application backend, serving the purpose of a database which automatically maintains the state of materialized views, accelerating reads. The maintenance is achieved through the use of a streaming dataflow graph which calculates and propagates changes to the materialized views. It is also efficient, acting like a cache that maintains only popular values of those materialized views. Other values are discarded and recomputed if requested, through the use of an *upquery*, a query going from the materialized view to the base tables. This is what is understood by Noria's partially-stateful dataflow model. Noria's original implementation did not concern itself with fault-tolerance, meaning that the materialized views would be recomputed in the case of failures, from the requests. In[58], Noria is extended with a unique form of causal logging, providing it with the ability to recover a single node while maintaining its materialized views intact. Noria achieves this through the use of *tree-clocks*, an elegant datastructure that tracks the provenance of records throughout the dataflow graph. In a sense, these are like vector clocks, for dataflows. By associating a tree-clock with each message delivered, operators can track the order in which all messages in the system were delivered. This information is efficiently shared with downstream operators by sending only the relevant parts of the tree-clock, the neighbourhood. As explained, in Section 2.2.2.4, downstream operators can then guide the recovery of failed upstream operators.

However, this implementation presents a few limitations. First, in Noria's case, only stateless deterministic operators are supported. This is because stateful streaming operators, like windowed aggregations, introduce other forms of nondeterminism, which tree-clocks cannot handle. The same goes for any nondeterminism that an operator could hold. Second, only one concurrent failure is supported, and no more than one successive failure is supported. It also requires a central coordinator. Noria's recovery uses a central controller to collect all differentials and tree-clocks, these are then used to solve an algebraic constraint problem, which yields the order in which the failed operator should deliver their messages and which messages they should not emit. Finally, no mention is found on the failures of sink operators, the last in the graph. These operators have no downstream and thus need special attention.

Ray[46] is a distributed application runtime targeting reinforcement learning and other AI tasks. Its programming model is based on tasks and actors, and actors, while the underlying execution engine is purely based on tasks. To provide fault-tolerance, the lineage of these tasks is tracked. Since actors are stateful entities, lineage graphs are augmented with state edges, which order task executions on the same actor. Originally, lineage tracking was done synchronously, much like Dryad or Spark. In [55], Ray was extended with the lineage stash, a technique that allows lineage to be logged asynchronously. If only asynchronous writing of the lineage was done, then in the case of failures, orphan tasks would be created. This is why the system was also extended with causal logging. When a task invokes a second task, it piggybacks the delta of the lineage graph. This way, if the first task fails before logging the lineage in stable storage, the second task is able to provide it with the necessary lineage information. However, much like Noria, the authors assume deterministic tasks and actors, meaning that again the only source of nondeterminism is the order of execution of tasks.

2.3.3 High-availability for Distributed Stream Processing

Some strategies for providing high availability in stream processing have been explored in the past with moderate success. In [37], the authors explore the use of classic *process-pairs* high-availability can be applied to stream processing, then move on to present their own solution.

Process-pairs high-availability involves having a primary and a secondary copy of an operator running simultaneously. The secondary may be in *passive-standby*, where it synchronously receives the operator state snapshots or in *active-standby*, where it receives and silently processes all records the primary receives. Under failure, the secondary takes-over execution and some recovery approach is used to bring the system back to a stable state. The authors highlight three categories of such approaches, one of which then subdivides into three cases:

- **Gap recovery:** Recovery methodology under which there may be a gap in outputted records after recovery.
- **Rollback recovery:** Recovery methodology under which the result allows duplicate records but is "equivalent" to a failure-free execution. This is because sources of nondeterminism may affect the results computed by the secondary operator. This is not to be confused with the Rollback Recovery techniques discussed earlier. Three subcases are identified:
  - **Repeating:** Duplicate records are identical to those produced previously by the primary.
  - **Convergent:** Duplicate records may be different, but the execution converges to the same state the failed operator was in.
  - **Divergent:** Duplicate records may be different and the execution diverges from the state the failed operator was in.
- **Precise recovery:** A recovery methodology under which failures are perfectly masked.

They also classify operators according to how they affect recovery semantics. The classes are, from most to least general, *arbitrary*, *deterministic*, *convergent-capable* and *repeatable*. Arbitrary operators have no constraints, deterministic operators apply a deterministic function on their inputs and state, convergent-capable operators can build their state from an empty state (e.g. BSort, aggregate) and reach a valid state, finally repeatable operators (map, filter) are convergent-capable operators whose duplicate records are exactly equal to the original.

The authors then conclude that passive standby is capable of delivering repeating rollback recovery for deterministic recovery and divergent rollback recovery for nondeterministic ones by having the recovered operator request replay of the latest messages. Importantly, the authors note that to achieve precise recovery with passive standby, operators must wait for a checkpoint until they are allowed to forward records downstream. We believe this overlooks the possibility of using causal logging techniques in order to achieve precise recovery. Active standby is classified the same way, with the exception that to provide precise recovery the primary node must synchronously send the determinants of its execution to the secondary. The authors also introduce their own solution, upstream backup, described earlier in Section 2.3.2.2.

2.3.4 Exactly-once delivery

As streaming systems converge on a distributed snapshotting for consistency, achieving exactly-once processing requires only restoring the graph to a previous point in time and a replayable source of input. Exactly-once processing is a requirement for exactly-once delivery. To provide exactly-once delivery two methods have been explored in the past [12, 19]:

- **Idempotent sinks**: If the pipeline is deterministic (unlikely, as this requires no message ordering delivery nondeterminism) and the sink is idempotent, then to achieve exactly-once delivery the sink merely has to eagerly push its results. If the pipeline is not deterministic then there additionally must be a write-ahead-log (WAL) of the operations to commit. When a checkpoint is completed, that epochs operations may be flushed from the WAL to the data sink. The idempotency guarantees that even if the flush is interrupted, it can be simply repeated to achieve exactly-once delivery.

- **Transactional sinks**: By coordinating the output of the sinks with the checkpointing mechanisms, using an algorithm like 2-phase commit [16], one can achieve exactly-once delivery. The implementation of the popular Kafka sink works as follows: on each epoch, each sink opens a transaction with kafka, into which records are streamed; when a new epoch starts the previous one is considered pre-commited; when notified of the completion of the first checkpoint, the pre-commited checkpoint is committed.

Both these methods increase latency as in-order to insure *read-committed* isolation, one must wait for the checkpoint notifications in order to read the new records. Depending on the checkpoint frequency this may add a lot to the latency.

2.3.5 Stream Provenance

Fine-grained lineage refers to tracking the path of records in a streaming computation. Thus the provenance of a record is the set of records of the previous streams used to compute this record.This information can be useful in debugging or reproduction of results. This section of related work is lightly covered, as it covers a lot of similar ground, such as operator instrumentation and piggybacking of encoded metadata.

Three identifiable methods exist for this. *Inversion*[57] implements this by having operators implement invertible functions. To compute provenance, an output record is inserted into the sink and flows backwards through the system. Unfortunately, most interesting functions are not invertible.

The second approach, *query rewrite*[33], implements this by creating two graphs. One computes the actual requested computation, the other, of the same shape tracks the lineage. The drawback of this approach is its overhead.

In Ariadne[34], the authors provide a third approach: *operator instrumentation*. Here, operators are instrumented with the ability to generate, propagate and delete provenance information from records. Records are augmented with provenance metadata and flow through the system carrying it. This is interesting because it is similar to the flow of determinants used in causal logging. The authors go through a few encoding optimizations which are relevant for this work:

- **Interval encoding**: Interval encoding is most useful in encoding tumbling windows, windows where the slide is larger than the size. The range of inputs is encoded instead of sending the full data. For example instead of sending the sequence 5,6,7,8,9, one may instead send 5-9.
- **Delta encoding**: Delta encoding is most useful for sliding windows with a small slide and large size. The difference from the previous window is encoded. As an example, take the window from the previous example as the past window, and 6,7,8,9,10 as the new window. We may encode simply the following -5,+10.
- **Dictionary compression**: Dictionary compression uses compression techniques to reduce the size of the data sent. This is useful above a certain threshold which may be found experimentally.

2.4 Conclusions

We observe that as streaming systems have evolved a few requirements have become mandatory. Early systems were very concerned with high-availability, but it was provided either at the cost of strong semantics or high overhead and computational requirements. Later systems have focused on providing strong semantics under failures, while giving up research on high-availability. Recent use-cases have revived the search for effective methods for high-availability, while maintaining these semantics. For this, a process-pairs approach can be taken, but is not enough to guarantee consistency.

Exactly-once processing and delivery semantics are extremely important to both simplify the lives of users and allow streaming systems to be used for reactive applications. This is highlighted by two facts: load-shedding techniques have lost popularity, while the use of message queues to store the streams in a stable fashion have increased in popularity. The use of message queues allows for pull-based sources to reset their position to a previous one in order to guarantee at-least-once processing.

Dataflow systems can be seen as assynchronous message passing systems, thus rollback recovery algorithms can be applied freely. Streaming systems already apply checkpointing to bound how far back in the stream they have to recover from. However, to maintain consistency, they are forced to rollback every node in the computational graph. Adding logging to these systems would allow a single node to be rolled back, recovering it from the logged determinants. Pessimistic logging is not a good choice as it requires accessing stable storage after processing every record, while optimistic logging risks losing determinants after a failure. Causal logging emerges as the solution, given that it can provide safety without requiring access to stable storage, and has also be shown to not have high overhead. The structured communication graph of streaming computations allows us to take advantage of several encoding tricks such as the ones used from stream provenance which may aid in further reducing the overhead of piggybacked data.

The use of transactional sinks allows for additionally guaranteeing at-most-once delivery under failures, which together with replayable sources provides exactly-once delivery, but increases latency due to batching of output commit. Causal logging would allow for operators to independently output commit while guaranteeing exactly-once delivery without the use of transactions, by attaching the determinants to the message to be sent.

Chapter 3

Proposed Work

This chapter begins by summarizing the proposed approach and highlighting the requirements of this new approach, as well as the improvements it enables to be developed on top of it. Following, the choice of framework on which this prototype will be developed is explained and further detail on it is given. Afterwards, the evaluation plan for this approach is proposed. The work to be performed is then decomposed into tasks, which are then scheduled in the remaining time of the thesis work.

3.1 Description

The proposed work involves designing, implementing and evaluating a fault-tolerance solution providing granular recovery of single operators to stream processors, while guaranteeing consistency after recovery and exactly-once processing guarantees, with low overhead. The solution proposed would use checkpointing along with causal logging to remove the nondeterminism present in between checkpoints. During failure-free operation operators would keep a volatile log of the messages they have sent since the last checkpoint, called an in-flight log. They also keep a volatile log of determinants, where they store both their determinants and all upstream determinants. Determinants are piggybacked on records sent. Checkpoint notifications trigger a truncation of these logs, bounding their size. If determinants are sent only one operator downstream, this provides a k-safety of one. This extends for any k. The extra overhead of tracking causality is of course expected to have an impact on throughput, however we believe the benefits outweigh the detriments in many use-cases.

To support a causal logging solution, one must verify the piecewise deterministic assumption, and thus one must correctly identify and log all sources of nondeterminism. Until now, identified sources are: message delivery order; accessing state external to the operator (e.g. a REST operation); the use of random number generation; the use of physical clock time and the elements inside a window at the moment it fires. Interestingly, punctuation such as watermarks, if logged along side records in the in-flight log, would make event-time windows deterministic.

During recovery of a failed operator d, d' the recovering operator first obtains the last checkpoint of d from the coordinator, then requests a copy of its determinants from downstream. It then requests replay of the messages since the checkpoint from its upstream nodes. Using the determinants it is able to deliver messages in the correct order and restore itself to the before failure state. Computation resumes as normal. Figure 3.1 shows a high-level view of this algorithm, where message delivery order is assumed to be the only source of nondeterminism. The in-flight log of a shows that it sent operator b the message v_1^a, while the determinant log of f shows that it knows that d delivered a message from b, then two messages from c.

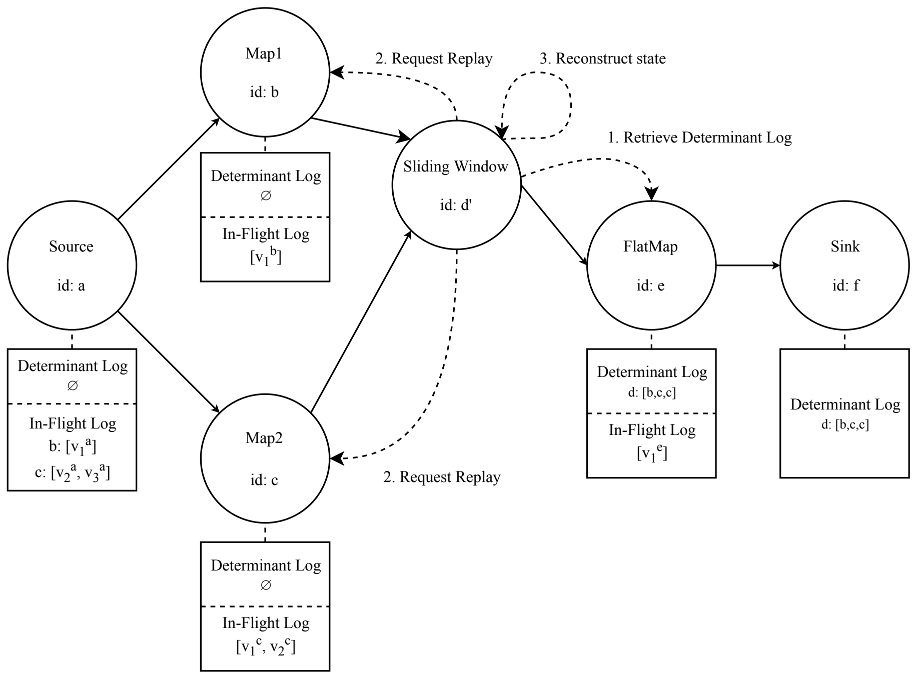

Figure 3.1: High-level view of the execution of the recovery algorithm.

Although the algorithm may seem simple, it is not trivial to operationalize, as stream processors are highly optimized systems, and any interference in their critical path may introduce bottlenecks. However implementing this type of fault-tolerance would allow a stream processor to evolve to provide more functionalities and guarantees:

- **High availability**: No current system supports extremely fast inexpensive consistent fail-over of failed operators. By combining this approach with an Active/Standby approach, a fast failover approach could be achieved where standby operators are quickly switched for failed operators and then the recovery algorithm is applied. Ideally, a standby could be active for several operators at a time.

- **Low latency exactly-once delivery**: Current streaming systems use transactions to commit output exactly once. This essentially batches the output leading to high latency. By storing the determinants in the external system to which we output, it would be possible to implement a recovery mechanism which ensures exactly-once delivery, but can continually output records.
- **Time travel debugging**: Having in-flight messages, and deterministic replayability, it is possible to have the ability to set an offset in a stream and look at the state in which it was, without needing to replay the stream. One must only use the determinants and in-flight messages to move the computations state forwards and backwards.
- **Uncoordinated checkpointing**: While coordinated checkpointing is necessary in current approaches, with causal logging we may choose different frequencies for checkpointing. For example, we may heuristically choose to checkpoint when the operator has the least state. Additionally, communication induced checkpointing may offer further optimizations.

3.1.1 Requirements

In this section the requirements of the solution are formalized using the MOSCOW method[56].

1. **Must have**: Support for fine-grained recovery of operators through the use of causal logging.
2. **Must have**: Consistent global state upon recovery.
3. **Must have**: Exactly-once processing guarantees.
4. **Must have**: Precise High availability.
5. **Must have**: An extensible determinant logging and replay method, to accommodate for new sources of nondeterminism.
6. **Should have**: Low latency exactly-once delivery as described in the above section.
7. **Should have**: Minimal effect on the throughput and latency of the system during failure-free operation.
8. **Should have**: Configurability, providing a trade-off between higher failure-free throughput and fault-tolerance guarantees.
9. **Should have**: Minimal global coordination, allowing for unaffected branches of computation to continue during recovery.
10. **Won't have**: Time-Travel Debugging.
11. **Won't have**: Uncoordinated checkpointing.

3.2 Framework for prototype

The framework chosen for implementing the prototype of this approach is Apache Flink[13]. The choice is based on several factors, first it is an open-source system, allowing for modifications. Second, it has a very strong community paired with a straightforward method for submitting improvements, allowing for possible contribution to a popular open-source project. It is also a streaming system built for real-time analytics, which will benefit from faster recovery. Several companies have already implemented large computations on Flink, in which case fast recovery becomes even more important. Interest in implementing more arbitrary computations in Flink is also rising, with some implementing social network backends[26], others using it as a backend for stateful functions as a service[2, 42, 49] and even general interest in the Kappa architecture[44]. These use-cases, being more Online Transaction Processing oriented, require fast processing of input with exactly-once delivery semantics, which will benefit from our approach. Unlike others, Flink already implements incremental and asynchronous checkpointing[20], and transactional exactly-once delivery [12], allowing for easier comparisons between approaches. It also integrates with several external systems, allowing plenty of choice in testing infrastructure.

3.3 Evaluation of solution

To evaluate the solution produced, we shall compare the system to an unmodified version of the system. Several metrics should be measured: the effect on throughput, the effect on latency and the additional memory overhead. Recovery time should also be measured and compared to the current solution. Correctness is also important to experimentally evaluate in such a complex recovery mechanism, meaning that the recovered state should be consistent with the pre-failure state. Several experimental parameters can already be identified, such as the level of paralelism and depth of the computation, as well as the state size of operators and checkpointing frequency. On a later phase, if time allows it, we could also explore the ability of this system to make progress in high churn environments such as edge environments or cheap preemptable computing cloud offerings.

3.4 Scheduling

To organize this work, we create four main goals: Design, Implementation, Evaluation and Reporting. Each of these goals is then broken up into granular tasks, easier to manage. Though some work on designing the algorithm must still be performed, as well as designing optimizations, some necessary components are already known to be needed and others can be estimated. The tasks and their scheduling are shown below:

1. Design
    a) Explore the existing code base.
    b) Find and document possible optimizations.
    c) Produce a design document detailing the following implementation.
2. Implementation
    a) Implement logging mechanisms.
    b) Implement recovery algorithm.
    c) Implement API to configure recovery mechanism.
    d) Implement low latency exactly-once delivery.
3. Evaluation
    a) Perform throughput, latency and memory usage benchmarks.
    b) Evaluate recovery time.
    c) Evaluate ability to make progress under failure.
4. Reporting
    a) Write thesis body.

Table 3.1: Scheduling proposed for thesis work.

Months	March				April				May				June				July				August				September			
Weeks	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th
Goal 1 - Design	X	X																										
Task 1.a	X																											
Task 1.b	X																											
Task 1.c		X																										
Goal 2 - Implementation			X	X	X	X	X	X	X	X	X	X	X	X														
Task 2.a			X	X	X	X																						
Task 2.b							X	X	X	X																		
Task 2.c											X																	
Task 2.d												X	X	X														
Goal 3 - Evaluation															X	X	X	X	X	X								
Task 3.a															X	X	X											
Task 3.b																		X										
Task 3.c																			X	X								
Goal 4 - Reporting															X	X	X	X	X	X	X	X	X	X	X	X	X	X
Task 4.a															X	X	X	X	X	X	X	X	X	X	X	X	X	X

Bibliography

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. “The design of the borealis stream processing engine.” In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289.

[2] A. Akhter, M. Fragkoulis, and A. Katsifodimos. “Stateful functions as a service in action.” In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 1890–1893.

[3] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. “MillWheel: Fault-Tolerant Stream Processing at Internet Scale.” In: *Very Large Data Bases*. 2013, pp. 734–746.

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. “MillWheel: fault-tolerant stream processing at internet scale.” In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1033–1044.

[5] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.” In: (2015).

[6] L. Alvisi. *Understanding the message logging paradigm for masking process crashes*. Tech. rep. Cornell University, 1996.

[7] L. Alvisi, K. Bhatia, and K. Marzullo. “Causality tracking in causal message-logging protocols.” In: *Distributed Computing* 15.1 (2002), pp. 1–15.

[8] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain, and A. De Mel. “An analysis of communication induced checkpointing.” In: *Digest of Papers. Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 99CB36352)*. IEEE. 1999, pp. 242–249.

[9] L. Alvisi, B. Hoppe, and K. Marzullo. “Nonblocking and orphan-free message logging protocols.” In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE. 1993, pp. 145–154.

[10] L. Alvisi and K. Marzullo. “Trade-offs in implementing causal message logging protocols.” In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. Citeseer. 1996, pp. 58–67.

[11] L. Alvisi and K. Marzullo. “Message logging: Pessimistic, optimistic, causal, and optimal.” In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 149–159.

[12] *Apache Flink exactly-once implementation*. https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html. Accessed: 2019-11-04.

[13] *Apache Flink frontpage*. https://flink.apache.org/. Accessed: 2019-11-04.

[14] M. D. de Assuncao, A. da Silva Veith, and R. Buyya. “Distributed data stream processing and edge computing: A survey on resource elasticity and future directions.” In: *Journal of Network and Computer Applications* 103 (2018), pp. 1–17.

[15] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. “Fault-tolerance in the Borealis distributed stream processing system.” In: *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. ACM. 2005, pp. 13–24.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley New York, 1987.

[17] B. H. Bloom. “Space/time trade-offs in hash coding with allowable errors.” In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[18] W. Brackenbury, R. Liu, M. Mondal, A. J. Elmore, B. Ur, K. Chard, and M. J. Franklin. “Draining the data swamp: A similarity-based approach.” In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2018, p. 13.

[19] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. “State management in Apache Flink®: consistent stateful distributed stream processing.” In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1718–1729.

[20] P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. “Lightweight asynchronous snapshots for distributed dataflows.” In: *arXiv preprint arXiv:1506.08603* (2015).

[21] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. “Apache flink: Stream and batch processing in a single engine.” In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[22] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. “Monitoring streams: a new class of data management applications.” In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 215–226.

[23] K. M. Chandy and L. Lamport. “Distributed snapshots: Determining global states of distributed systems.” In: *ACM Transactions on Computer Systems (TOCS)* 3.1 (1985), pp. 63–75.

[24] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik. “Scalable Distributed Stream Processing.” In: *CIDR*. Vol. 3. 2003, pp. 257–268.

[25] J. Dean and S. Ghemawat. “MapReduce: Simplified data processing on large clusters.” In: (2004).

[26] *Drivetribes Flink Backend*. `https://www.ververica.com/blog/drivetribe-cqrs-apache-flink`. Accessed: 2019-11-04.

[27] E. N. Elnozahy and W. Zwaenepoel. “Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit.” In: *IEEE Transactions on Computers* 5 (1992), pp. 526–531.

[28] E. N. Elnozahy. “Manetho: fault tolerance in distributed systems using rollback-recovery and process replication.” Doctoral dissertation. Rice University, 1994.

[29] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. “A survey of rollback-recovery protocols in message-passing systems.” In: *ACM Computing Surveys (CSUR)* 34.3 (2002), pp. 375–408.

[30] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey. “DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language.” In: *Proc. LSDS-IR* 8 (2009).

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung. “The Google file system.” In: (2003).

[32] J. Gjengset, M. Schwarzkopf, J. Behrens, L. T. Araújo, M. Ek, E. Kohler, M. F. Kaashoek, and R. Morris. “Noria: dynamic, partially-stateful data-flow for high-performance web applications.” In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 213–231.

[33] B. Glavic and G. Alonso. “Perm: Processing provenance and data on the same data model through query rewriting.” In: *2009 IEEE 25th International Conference on Data Engineering*. IEEE. 2009, pp. 174–185.

[34] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul. “Ariadne: Managing fine-grained provenance on data streams.” In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 39–50.

[35] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. “Graphx: Graph processing in a distributed dataflow framework.” In: *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 2014, pp. 599–613.

[36] J. Hamilton. *The cost of latency*.

[37] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. “High-availability algorithms for distributed stream processing.” In: *21st International Conference on Data Engineering (ICDE’05)*. IEEE. 2005, pp. 779–790.

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. "Dryad: distributed data-parallel programs from sequential building blocks." In: *ACM SIGOPS operating systems review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[39] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru, E. Johnson, M. Spicer, and A. E. Sariyüce. "Consistent regions: Guaranteed tuple processing in ibm streams." In: *Proceedings of the VLDB Endowment* 9.13 (2016), pp. 1341–1352.

[40] D. B. Johnson and W. Zwaenepoel. *Sender-based message logging*. Rice University, Department of Computer Science, 1987.

[41] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. "Benchmarking Distributed Stream Data Processing Systems." In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018). DOI: 10.1109/icde.2018.00169. URL: http://dx.doi.org/10.1109/ICDE.2018.00169.

[42] A. Katsifodimos and M. Fragkoulis. "Operational Stream Processing: Towards Scalable and Consistent Event-Driven Applications." In: 2019.

[43] L. Lamport. "Time, clocks, and the ordering of events in a distributed system." In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[44] J. Lin. "The lambda and the kappa." In: *IEEE Internet Computing* 5 (2017), pp. 60–66.

[45] F. Mattern et al. *Virtual time and global states of distributed systems*. Citeseer, 1988.

[46] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, et al. "Ray: A distributed framework for emerging {AI} applications." In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 561–577.

[47] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. "Pig latin: a not-so-foreign language for data processing." In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1099–1110.

[48] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. "The hadoop distributed file system." In: *MSST*. Vol. 10. 2010, pp. 1–10.

[49] *Statefun*. http://statefun.io.

[50] R. Strom and S. Yemini. "Optimistic recovery in distributed systems." In: *ACM Transactions on Computer Systems (TOCS)* 3.3 (1985), pp. 204–226.

[51] Y. Tamir and C. H. Séquin. "Error Recovery in Multicomputers Using Global Checkpoints." In: *In 1984 International Conference on Parallel Processing*. 1984, pp. 32–41.

[52] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. "Hive: a warehousing solution over a map-reduce framework." In: *Proceedings of the VLDB Endowment* 2.2 (2009), pp. 1626–1629.

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

[54] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. "Exploiting punctuation semantics in continuous data streams." In: *IEEE Transactions on Knowledge and Data Engineering* 15.3 (2003), pp. 555–568.

[55] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. "Lineage stash: fault tolerance off the critical path." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM. 2019, pp. 338–352.

[56] K. Waters. "Prioritization using moscow." In: *Agile Planning* 12 (2009), p. 31.

[57] A. Woodruff and M. Stonebraker. "Supporting fine-grained data lineage in a database visualization environment." In: *Proceedings 13th International Conference on Data Engineering*. IEEE. 1997, pp. 91–102.

[58] G. Yuan. "Scalable Fault Tolerance for High-Performance Streaming Dataflow." Doctoral dissertation. Massachusetts Institute of Technology, 2019.

[59] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2.

[60] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster computing with working sets." In: *HotCloud* 10.10-10 (2010), p. 95.

[61] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. "Discretized streams: Fault-tolerant streaming computation at scale." In: *Proceedings of the twenty-fourth ACM symposium on operating systems principles*. ACM. 2013, pp. 423–438.

[62] S. Zeuch, A. Chaudhary, B. Del Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Breß, J. Traub, and V. Markl. "The NebulaStream Platform: Data and application management for the internet of things." In: *arXiv preprint arXiv:1910.07867* (2019).