Scalable and Reliable Data Stream Processing

PARIS CARBONE

Doctoral Thesis in Information and Communication Technology
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2018

School of Electrical Engineering
and Computer Science
KTH Royal Institute of Technology
SE-164 40 Kista
SWEDEN

TRITA-EECS-AVL-2018:54
ISBN: 978-91-7729-901-1

Akademisk avhandling som med tillstånd av Kungliga Tekniska Högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i informations- och kommunikationsteknik på fredagen den 28 september 2018 kl. 09:00 i Sal A, Electrum, Kungliga Tekniska Högskolan, Kistagången 16, Kista.

Printed by Universitetsservice US-AB

To Mary and Nathalie

Abstract

Data-stream management systems have for long been considered as a promising architecture for fast data management. The stream processing paradigm poses an attractive means of declaring persistent application logic coupled with state over evolving data. However, despite contributions in programming semantics addressing certain aspects of data streaming, existing approaches have been lacking a clear, universal specification for the underlying system execution. We investigate the case of data stream processing as a general-purpose scalable computing architecture that can support continuous and iterative state-driven workloads. Furthermore, we examine how this architecture can enable the composition of reliable, reconfigurable services and complex applications that go even beyond the needs of scalable data analytics, a major trend in the past decade.

In this dissertation, we specify a set of core components and mechanisms to compose reliable data stream processing systems while adopting three crucial design principles: *blocking-coordination avoidance, programming-model transparency*, and *compositionality*. Furthermore, we identify the core open challenges among the academic and industrial state of the art and provide a complete solution using these design principles as a guide. Our contributions address the following problems: I) *Reliable Execution and Stream State Management*, II) *Computation Sharing and Semantics for Stream Windows*, and III) *Iterative Data Streaming*. Several parts of this work have been integrated into Apache Flink, a widely-used, open-source scalable computing framework, and supported the deployment of hundreds of long-running large-scale production pipelines worldwide.

Sammanfattning

System för strömmande databehandling har länge ansetts vara en lovande arkitektur för snabb datahantering. Paradigmen för strömmande datahantering utgör ett attraktivt sätt att utrycka tillståndbaserad persistent tillämpningslogik över evolverande data. Men trots många bidrag i programmeringssemantik som adresserar vissa aspekter av dataströmning, har befintliga tillvägagångssätt saknat en tydlig universell specifikation för den underliggande systemexekveringen. Vi undersöker system för strömmande databehandling som en generell skalbar beräkningsarkitektur för kontinuerliga och iterativa tillämpningar. Dessutom undersöker vi hur denna arkitektur kan möjliggöra sammansättningen av pålitliga, omkonfigurerbara tjänster och komplexa tillämpningar som går utöver behoven av den förnärvarande trendiga BigData-analysen.

I den här avhandlingen specificerar vi en uppsättning kärnkomponenter och mekanismer för att sätta samman tillförlitliga system för strömmande databehandling. Samtidigt antar man tre viktiga konstruktionsprinciper: undvikandet av blockerande samordning, transparens av programmeringsmodellen, och sammansättningsbarhet. Vidare identifierar vi de huvudsakliga öppna utmaningarna inom akademi och industri i området, och föreslår en fullständig lösning med hjälp av de ovan nämnda principerna som guide.Våra bidrag adresserar följande problem: I) Tillförlitlig exekvering och tillståndhantering för dataströmmar, II) delning av beräkningar och semantik för Ström Windows, och III) Iterativa dataströmmar. Flera delar av detta arbete har integrerats i Apache Flink, ett allmänt och välkänt beräkningsramverk, och har använts i hundratals storskaliga produktionssystem över hela världen.

Acknowledgements

Throughout my PhD studies I had the pleasure to meet and collaborate with people from various research groups and open source development communities as well as making good friends on-campus and beyond. I consider all these people fellow companions in this doctoral journey and every and each interaction I had with them an indirect contribution to this work.

First and foremost I would like to express my deep gratitude to my main advisor, Seif Haridi that mentored me with professional insightfulness and wisdom all these years and to whom I attribute the most broad spectrum of knowledge I acquired during that time. I would also like to thank my external advisor, Asterios Katsifodimos for the amount of energy and exceptional coaching he put to encourage positive and creative thinking and make my ideas appealing to the database research community (a nearly impossible task). In addition, I would like to thank the rest of my university mentors: Christian Schulte, my quality assurance reviewer and doctoral studies advisor for his sharp and always accurate advice as well as Jim Dowling and Vladimir Vlassov, my secondary supervisors for exposing me to interesting open problems and topics at the earliest stages of my studies. I also want to thank Prof. Peter Pietzuch for expressing his interest to be the opponent of my thesis and Professors Yanlei Diao, Marianne Winslett and Mads Dam for their interest to take part in my doctoral defense committee, it is truly an honor.

I consider my second PhD year a core landmark in my career so far since that was the year I joined the Apache Flink community (called Stratosphere back then). This gave me the chance to apply in a real setting many principles of distributed computing I studied previously as well as communicate my work to a much broader audience. None of my efforts would make an impact if not embraced and reinforced by Stephan Ewen (project VP), Gyula Fóra, Vasia Kalavri and Kostas Tzoumas with whom I interacted the most during my PhD over endless whiteboard brainstorming or late night discussions accompanied by Club-Mate and German beer. I also thank the broader Flink community for their hard work, friendship and support, especially Theo, Ufuk, Stefan, Aljoscha, Kostas K., Max, Marton and Robert among fellow "squirrels" as well as affiliated partners and friends for their insights such as Frank McSherry, Volker Markl, Tilmann Rabl, Tyler Akidau, Flavio Junqueira, Gianmarco De Francisci Morales, Pramod Bhatotia and Alexander Alexandrov.

Among my Stockholm friends and on-campus colleagues Lars Kroll has been the closest one from my very first day in Sweden. I never regretted any of the sailing mornings, pipe smoking evenings or whiskey nights we had together. In fact, it was during one of those times that he convinced me to start a PhD. My time with Lars has always been a true source of rich scientific, sometimes musical and often deep philosophical inspiration. Furthermore, I am also grateful of our other "triumvirate" member, Alex Ormenisan, a hard-working researcher, expert role playing gamer and great friend with a nerdy temperament that made each day in the office feel like an episode of Big Bang Theory.

I also had the chance to supervise closely many exceptional MSc students that contributed to my work such as Jonas Traub (Window Semantics), Daniel Bali (Graph Streams), Fay Beligianni (Stream ML), Martha V. Konchylaki (Stream Sampling), Marius Melzer (Iterative Progress Tracking) and Zainab Abbas (Graph Stream Partitioning). Same applies to our latest batch of students that contributed to the fundamental design of our upcoming ambitious system, namely Klas Segeljakt, Johan Mickos, Oscar Bjur and Max Meldrum.

I am particularly grateful for Sandra Nylén, our department's charismatic secretary as well as as Madeleine Printzsköld and Thomas Sjöland our department's HR and Head Manager for helping me out instantly when in need. Same applies to Sverker Jansson, head of the CSL lab at the Swedish Institute of Computer Science (SICS), who encouraged me to participate in all great discussions and events of SICS. I would also like to extend my appreciation to all KTH and SICS friends for the joint lunches, paper reading groups and fika[1], namely Ahmad, Amir, Anis, Benoit, David, Daniel, Fatemeh, Gabriel, Hooman, Jingna, Johan, Kamal, Mahmood, Martin, Niklas, Roberto, Romy, Salman, Saranya, Sarunas, Shatha, Stelios and Ying. Finally, I would also like to express my gratitude to the Swedish Foundation for Strategic Research (SSF) for funding most of my doctoral research.

Several influencers back from my undergraduate studies also inspired scientific excellence in me and set a great example throughout their teachings. Among others, I would like to thank Andreas Veneris, Vasilis Vassalos, Christos Papadimitriou, Ioannis Kontoyiannis, Vana Kalogeraki, George Xylomenos and Giannis Kareklas, as well as Aris, Katerina and Anna for their support during those early times.

Last but not least, the full credit of my work goes to my wife Mary and our baby daughter Nathalie for their noteworthy patience and endurance throughout my frequent trips and ridiculously late working hours. Apart from her unmatched love and support, Mary always gave me the data analyst's perspective of things to consider when designing systems as well as teaching me how to optimize my time and be efficient, a native skill of hers. I am grateful of my family and I dedicate my past, current and future work to them.

[1]Swedish word for "coffee break"

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER

1

Introduction

Recent advances in distributed and cloud computing systems have driven the domain of data management into a "big data" era that has been known for critical paradigm shifts. A significant decrease in the cost of storage and compute resources as well as the inception of utility computing contributed to the mass development and adoption of scalable system designs. However, as with most technological transitions, intermediate states of the art often related to design choices with unnecessary trade-offs, attributed to the lack of broadness when it comes to potential application domains and the integration of a long history of interdisciplinary research findings.

Map-Reduce, has been a prominent transition step for data management since it transformed bulk data processing workloads from a monolithic database query execution to a distributed job execution model, appropriate for share-nothing system architectures. Despite its heavy focus on scalability and fault tolerance, Map-Reduce had been criticized for employing inconvenient limitations, particularly on enforcing staged execution between every computational step as well as limiting programs to acyclic computational graphs of stateless tasks. The latter made iterative programs such as interactive query execution, machine learning and graph processing workloads inaccessible without any form of data sharing between tasks and opened up a domain of special-purpose scalable systems addressing some of these limitations. Apache Spark can be considered an immediate successor of Map-Reduce-related technologies since it complemented existing core design properties such as scalability and fault tolerance while overcoming programming model limitations which existed purely as design choices. Namely, Spark's adoption of lineage graphs showcased that data sharing can be achieved at no additional cost in a distributed architecture, while stages can make use of in-memory replication instead of a distributed file system.

The next radical shift in data management, which is the main focus of this dissertation, has been the one of "scalable stream processing". Stream processing itself has its roots in *event-based systems* [1] such as publish-subscribe middleware as well as data stream management (DSMS) from the database research field. The differentiating characteristic of this system class is the notion of data itself, that is a *continuous*, possibly *infinite* resource instead of "facts and statistics organized and collected together for future reference or analysis" [1]. Pre-existing forms of stream processing can be observed within respective domains, such as network programming on byte streams, functional (e.g., monads) and actor programming, complex event processing and database materialized views. *Stream Management* had also been an active research field for decades [2, 3, 4]. Nonetheless, several of these ideas have been only just recently put consistently together to compose new software stacks (including storage, delivery, processing and domain-specific languages) for writing scalable applications centered around the notion of data as an unbounded resource.

Several popular open source developments in stream processing technology were incremental approaches that aimed primarily to offer a low-latency alternative to MapReduce for real-time processing which traded off reliability and expressivity power in order to prioritize for speed. Twitter's Storm system [5] is an example of a scalable data streaming system with limited guarantees that became popular around the time the work of this dissertation begun. Similarly, academic research related to stream processing focused heavily on approximate data structures [6] and specialized application domains (e.g, complex event processing), also contributing to the same misconception. These led to the creation of a dogma that labeled data streaming as an approximate technique for data analysis. The *lambda architecture* [7] proposal emphasized this idea by separating the concerns of reliable large-scale data processing and stream processing using a "batch" and "speed" layer respectively. Nevertheless, recent efforts -including work that is presented in this dissertation- revisited this technology and brought it back to the surface, not as a niche or complementary approach to the state of the art but a radically general proposition to data-centric, reliable programming, capable to subsume existing programming models (e.g., MapReduce [8]). In fact, stream processing has broadened rather than restricting the scope of data management from retrospective data analysis to continuous, unbounded and scalable processing coupled with persistent state.

In this thesis, we investigate a rational system design for tackling some of the most prominent open challenges in scalable data streaming: I) *Fault Tolerance and Scalable State Management*, II) *Computation Sharing and Semantics for Sliding Windows*, and III) *Iterative Data Streaming*. While partial solutions have been proposed for some of these problems in the past, instead, we seek for mechanisms that do not impose unnecessary performance trade-offs and separate runtime concerns from any user-facing programming model. Throughout this work, we encapsulate these

[1]Current definition of "data" according to Google Dictionary (2018)

requirements through a set of adopted design properties: *blocking-coordination avoidance*, *programming-model transparency*, and *compositionality*. Several parts of this work have been contributed to the core of Apache Flink, an open source stream processing system that has met with great success via its widespread adoption in the industry. Flink has so far been integrated and tested in hundreds of long-running large-scale production pipelines by world-class industrial companies including Alibaba, King, Uber, Zalando, Hwawei and Ericsson.

1.1 Open Challenges in Scalable Stream Processing

We identify three most critical, yet challenging open underlying problems observed in both academic and industrial system architectures for scalable data stream processing, sorted by their importance for adoption.

1.1.1 [C1] Fault Tolerance and Scalable State Management

Data streams can be arbitrarily long, having no clear beginning or end and thus, hint the requirement of unbounded computation. In the context of batch processing computation is strictly coordinated and staged to eventually finish and return back to the user with an output after its execution. However, during the lifetime of a stream processing application any user-declared logic can be potentially executed arbitrarily long, raising concerns regarding the size of any declared state and provided processing guarantees when partial failures occur such as crashes, disk and network failures. The non-trivial nature of this problem led to approaches that either omitted any form of state management, leaving all concerns about the lifecycle of application state to the user [5], or offloaded state handling to external data storage systems [9].

One of the main goals of this work is to provide a core execution model for stateful data stream processing, specify its properties including necessary processing guarantees and design underlying distributed algorithms that can fulfill those properties, in a rigorous, system-oriented approach. When it comes to programming abstractions, we investigate the case of making state explicit to the system [10, 11] for covering transparently a plethora of state management needs that involve reconfiguration and fault tolerance without making crucial sacrifices in terms of program expressibility. Finally, when it comes to performance, we seek for reliable execution mechanisms that do not impose blocking coordination contrary to other approaches that adopt strict synchronization barriers [12].

1.1.2 [C2] Computation Sharing and Semantics for Sliding Windows

Computation sharing can yield high performance benefits in long-running task execution and its applicability can be crucial when different parts of a continuous computation overlap such as the case of sliding windows. Stream windows make blocking operations such as finite set functions feasible within an unbounded stream. Windows are in fact one of the first exclusive primitives in data streaming which allowed ingested stream records to be grouped by some common characteristic. Furthermore, stream windows are not ordinary sets since they include the ability to evolve or "slide" over time while stream ingestion is ongoing. A common way to define sliding windows is by defining their effective size or "range" as well as their "slide" which encodes the frequency at which we want to re-execute our window computation that is typically some associative aggregation function. A very simple *count window* of range 100 and slide 5 will always contain the last 100 records ingested and trigger its associated computation upon every 5th record that is received.

One of the problems that arises in such a repeating computation is how to optimize both memory and computation in order to avoid extreme levels of redundancy (in this case 95 records are re-aggregated per consecutive window). At one extreme, the majority of production systems and Domain Specific Languages (DSLs) for data streaming today restrict stream windows to a limited class of periodic windows which can be trivially optimized using a technique called *slicing* [13, 14]. At the other extreme, general pre-aggregation data structures such as FlatFat [15] can be used for aiding computations on windows of arbitrary size and frequency. The latter allows support for special-purpose and user-defined windows at a high runtime complexity cost. In our study, we question the distinction between periodic and non-periodic windows and extend computation sharing further to more complex and interesting user-defined stream windows.

1.1.3 [C3] Iterative Stream Processing

Forms of iterative computation have a significant role in data analytics. For example, Bulk Synchronous Parallel (BSP) [16] processing has been popularized by large scale graph processing systems such as Pregel [17], Giraph [18] and most recently deep learning [19]. The same applies for fixpoint iterative approximations in online, large-scale machine learning where most algorithms require de-facto support for iterative processing primitives. However, with a few exceptions [20] such primitives have only been considered within short-running task executions such as batch processing systems where the implementation of iterative steps is strictly coordinated by nature.

In this work, we aim to examine low level primitives and non-blocking, decentralized iterative coordination protocols to incorporate native loops in stream processing applications. We therefore investigate the prospect of embedding cyclic progress metrics into already existent application-time progress tracking mechanisms such as low watermarking [21, 9, 22, 23] which are widely used in stream processing execution systems today for out-of-order processing.

1.2 Design Principles

Throughout this extended study we identify a set of system design principles that outline basic requirements of scalable data stream processing. These requirements are used to bolster many of our design choices as well as pinpointing the limitations of existing approaches to several of the challenges we are aiming to tackle.

[D1] Blocking-Coordination Avoidance: Certains forms of distributed coordination are paramount in the broad context of distributed computing systems. Distributed coordination is typically associated with any communication protocol executed across multiple distributed processes connected over a network to ensure that a global condition is satisfied. Examples can be found in distributed database transactions [24], shared memory and consensus for replicated state machines [25, 26] as well as master-driven execution of bulk synchronous iterative processing and staged computation in scalable data processing systems [27, 8]. While many distributed coordination problems can be solved strictly with *blocking synchronization*, disallowing processes to make progress until a condition is known to be satisfied, it is favorable to be avoided when that is feasible. Instead, in the context of latency-critical stream processing systems it is preferred to adopt non-blocking synchronization mechanisms which can allow uninterrupted computational progress while not violating necessary correctness guarantees. Throughout this text we will use the term "blocking-coordination avoidance" to refer to this property.

[D2] Runtime Transparency: A common challenge in scalable computing is to provide a clear separation of concerns between a system's execution and its programming interface. We notice two equally hazardous effects of non-transparency in various programming models. First, we often see heavy restrictions and crucial sacrifices in the expressive power of a programming model as a way to optimize execution for cases that are known to run optimal in a certain runtime. An example is the common restriction of sliding window semantics to periodic count and time windows that can be trivially aggregated using non-redundant methods such as slicing [14, 13]. Another transparency violation is to demand more input from the users than the logic of their program, such as configuration parameters or logic that is solely relevant to the runtime. An example of that case of transparency violation is the declaration of micro-batching discretization granularity in Apache Spark [12], the necessity of which is only relevant for integrating with that specific runtime. In this work, we aim to avoid both types of transparency violation as a way to offer a clear separation of concerns between the 'what' and 'how' in stream processing.

[D3] Model Compositionality: One of the most fundamental concepts in computer systems is the ability to compose arbitrarily complex logic and operations out of a small set of basic primitives. In stream processing compositionality relates to the ability of building applications with arbitrary state, possibly being able to create complex state representations out of primitive ones, or the ability to declare and run arbitrarily nested iterative logic. In this work, we aim to provide the basic building blocks that can cover all aspects of stateful event-based logic, user-defined windowing semantics and nested iterative logic.

To summarize, this work aims to seek and propose solutions to core challenges in scalable stream processing (C1-C3) while satisfying a set of crucial design principles (D1-D3). The aforementioned aims form the current work's thesis:

Thesis: *A reliable system design for continuous stateful stream processing at scale can support model transparency and compositionality without imposing blocking coordination.*

1.3 Primary Contributions

We summarize the main results of this work into the following contributions, each addressing challenges C1-C3 respectively.

1.3.1 Asynchronous Epoch Commits

We define a simple model for distributed stream execution that divides computation into epochs, each of which is completed atomically. After each epoch the complete state of the system is available for a plethora of state management needs such as fault recovery and reconfiguration. To capture complete epochs consistently, we define the concept of an *epoch cut* a special form of Chandy and Lamport's distributed consistent cut [28]. Furthermore, we solve the problem of acquiring the complete state defined by an epoch cut asynchronously without the need for blocking coordination. Our mechanism [29, 30] respects causal dependencies in cyclic or acyclic dataflow graphs and is transparent to the programming model, allowing arbitrary stateful event-based logic on persistent state (locally embedded or external). In addition, the same mechanism respects in-flight flow control mechanisms and leads to minimal snapshots, restricting in-channel logging to dataflow cycles. Among others, we show how epoch snapshots can be used in practice to support end-to-end guaranteed consistent processing, reconfiguration and application provisioning while enabling operations that were not considered feasible in the past such as external state querying under snapshot-isolation guarantees. Chapters 3 and 4 address the theoretical and practical part of this contribution respectively.

1.3.2 Shared Computation of User-Defined Windows

Within the scope of the Cutty Aggregator [31], we examine all known cases of sliding windowing aggregation and identify the minimum programming model requirements to achieve optimal shared execution. We encapsulate these requirements in the proposal of "Deterministic Windows", a new class of windowing semantics that expands largely beyond the popular, yet restricted, class of "Periodic Windows". Our Cutty aggregator exploits deterministic window properties to optimally perform pre-aggregation using a series of non-overlapping segments. Our technique further adapts circular heap data structures [15, 32] previously used for evaluating arbitrary aggregate lookups on raw streams, to sliced aggregates for reducing even further the final evaluation of complete window aggregates. Our complexity analysis and multi-query evaluation on Apache Flink showcases orders of magnitute of computational benefits that are now possible at a significantly lower memory footprint. Finally, our technique can multiplex well multiple diverse dynamic windows that can change or reconfigure at any time during the lifetime of an application while requiring zero semantical knowledge to be known apriori at compilation time. In practice, this means that our approach allows for a general programming model for window computation, transparent to the aggregation sharing strategy and the composition of applications with diverse windowing semantics. Chapter 5 offers a full description of the problem of window computation sharing and our solution.

1.3.3 Structured Loops and Window Iterations

In our approach, we first examine the concept of iterative processes and identify the universal set of programming primitives used to describe them. We then identify the challenges of implementing these primitives in a distributed architecture and present an extension of the out-of-order stream processing model for cyclic computation. We incorporate the complete logic needed to track cyclic progress within a special "structured loop" operator that includes the necessary additions to low-watermarking for nesting additional progress measures (other than application-time) and further allows for building structured programs using data streams. The decentralized progress tracking protocol and other execution concerns are hidden from the user-facing programming model, making it a highly transparent mechanism. As a proof of concept we have composed a multi-pass window aggregation framework on top of structured loops that can be used to model iterative processes on stream windows, as well as a vertex-centric framework on top. Finally, we discuss all implementation concerns related to stream iterations such as flow control and propose a future line of work to make stream cycles a standard component of modern stream applications. Chapter 6 summarizes the problem of stream iterations and our contributions.

1.3.4 Industrial Adoption and Other Contributions

Most parts of this work were made alongside contributions in the development of the Apache Flink system as well as external frameworks, namely Gelly-Streams and Apache Samoa an online graph and machine learning framework respectively.

Flink is now a leading open-source system in data stream processing with more than 300 contributors. Among research and other code contributions such as JIRA issues, the core findings and prototypes developed throughout this thesis were integrated to the core of Flink and included in major releases of the system:

- User-Defined Windows - 0.8.0 release
- Pipelined Snapshots and Exactly-Once Processing - 0.9.0 release
- Redesign of Stream Loops (FLIP-15 [2])

Gelly-Streams [3]is an open source library for writing graph-centric applications that operate on a streaming model. Gelly-Streams is built on Apache Flink and is currently maintained as an independent software library. Despite not being part of this dissertation, our involvement in the conception and development of this graph streaming framework inspired us to work on several of the main challenges and bolstered our research findings as a future-proof concept application. In the context of Apache Samoa, an engine-independent framework for scalable online machine learning, we contributed by developing a runner for Flink. By translating every stream machine learning library to a Flink dataflow execution graph we therefore enabled the prospect of running fault tolerant distributed machine learning tasks on data streams.

Adoption: Flink is currently serving the data processing needs of companies ranging from small startups to large enterprises and offered commercially via different vendors such as data Artisans (core maintainers of the framework), Lightbend, Amazon AWS, and Google Cloud Platform. The growing number of large-scale fault tolerant deployments of the system showcase the applicability and performance of our snapshotting algorithm. Among the largest deployments Alibaba, the world's largest e-commerce retailer company, maintains a 1000-node Flink deployment to support a critical search service [33] while Uber, the world's largest privately held company has built and deployed internally AthenaX a query platform based on SQL and Flink [34]. Other examples are King (owned by Activision Blizzard), the largest mobile gaming company at the time of writing, which has based their standing query infrastructure on top of Flink's flexible state management. Moreover, Netflix is building a self-serve, scalable, fault-tolerant, multi-tenant "Stream Processing as a Service" platform leveraging Apache Flink with the goal to make the stream of more than 3 petabytes of event data per day available to their internal users [35]. Other use cases come from the banking sector [36] and telecommunications [37].

Standardization: Beyond Flink, many communities have embraced the concept of distributed consistent snapshots for the purposes of continuous processing.

[2]https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=66853132
[3]https://github.com/vasia/gelly-streaming

Since the initial contribution of the snapshotting algorithm there have been several adaptations of our original work. Examples are Apache Storm's snapshotting by Hortonworks [38], Apache Apex's pipelined snapshots[39] as well as epoch-based execution in Apache Spark's latest Continuous Processing Mode [40]. Another aspect that shows the impact of our work is the modeling effort of stream snapshots [41], that was recently initiated by Google, from the team that worked on Millwheel [9] and Apache Beam/Google Dataflow [21, 42]. The creation of a standard for stream snapshots can further serve as a strong interoperability link between systems, enabling the migration of applications across arbitrary runners upon demand.

1.4 Previously Published Work

The majority of the content of this dissertation is based on previously published work that has been peer-reviewed in its majority. More specifically, Chapter 2 includes existing and in-part revised content from papers on Flink [43, 29] as well as a Springer book chapter surveying the state of the art in stream processing technology [44]. Most ideas and findings of Chapters 3 and 4 have been previously published in a prestigious IEEE special issue in Data Engineering as well as PVLDB [29, 43]. An older arXiv paper [30] further outlines many of the ideas presented in the same chapter. Furthermore, Chapter 5 is based on our work in Cutty which was published a ACM CIKM paper [31] and later revisited in a Springer book chapter analyzing sliding window aggregation and semantics [45]. Finally, the work on stream iterations is currently under submission.

The complete list of material published and presented in conferences, journals and chapters along with assigned contributions is the following:

Conference Papers

P1 Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. **"State management in Apache Flink®: Consistent Stateful Distributed Stream Processing."** Proceedings of the VLDB Endowment 10, no. 12 (2017): 1718-1729.

Contribution The author of this dissertation designed and prototyped the core snapshotting algorithm and led the design and writing of the paper.

P2 Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. **"Cutty: Aggregate Sharing for User-Defined Windows."** In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 1201-1210. ACM, 2016.

Contribution: The author of this dissertation designed and co-implemented the Cutty technique and conducted the evaluation as well as leading the design and writing of the paper.

Bulletin Issues

P3 Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. **"Apache Flink^{TM}: Stream and Batch Processing in a Single Engine."** Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36, no. 4 (2015).

Contribution: The author of this dissertation co-authored the article, and led the parts related to stateful processing, fault tolerance and iterations.

Book Chapters

P4 Paris Carbone, Gábor E. Gévay, Gábor Hermann, Asterios Katsifodimos, Juan Soto, Volker Markl, and Seif Haridi. **"Large-Scale Data Stream Processing Systems."** In Handbook of Big Data Technologies, pp. 219-260. Springer, Cham, 2017.

Contribution: The author of this dissertation co-authored the chapter, and led the writing of its Introduction and Systems sections.

P5 Paris Carbone, Asterios Katsifodimos, and Seif Haridi. **"Stream Window Semantics and Optimisation"** In Encyclopedia of Big Data Technologies, Springer, Cham, 2018.

Contribution: The author of this dissertation led the writing of the chapter.

ArXiv Articles

P6 Paris Carbone, Gyula Fóra, Stephan Ewen, Seif Haridi, and Kostas Tzoumas. **"Lightweight asynchronous snapshots for distributed dataflows."** arXiv preprint arXiv:1506.08603 (2015).

Contribution: The author of this dissertation led the writing of the article.

1.5 Research Methodology

The first step, prior to designing system algorithms and techniques is to identify the actual problem. Sections 1.1 and 1.2 discussed three major challenges (C1-C3) alongside desirable properties (D1-D3) in scalable and realiable stream processing.

We identified these requirements throughout surveying the research and industrial state of the art, having most parts of published in P4 [44] as a *literature review*. Within the same study we recognized where existing solutions fell short in respecting all desirable design properties and further motivate seeking complete solutions to these issues among our contributions.

Following the literature review, we present designs of system elements (programming model primitives, algorithms etc.) and prove that they fulfill the requirements of providing system-level support for each respective challenge C1-C3. For each of our proposed solutions, we adopted a selection of appropriate research methodologies underlined as follows:

- C1: The adoption of the 'Asynchronous Epoch Commit' protocol is first supported by a *"formal"* methodology, which is termed necessary to prove the satisfiability of critical properties. In addition to formal proofs, a *"build"* research methodology (prototype implementation) is used to demonstrate certain design properties in real system deployments that have otherwise not been demonstrated before. Finally an *"experimental"* methodology is adopted for evaluation phase of that study.
- C2: A more focused *"literature review"* of sliding window aggregation techniques (later published in P5) contributed to our general understanding of the core programming model primitives required for optimal system-level aggregation. Thus, we continue with a *"model"* methodology in the encapsulation of "deterministic user-defined windows" followed by a *"build"* method, developing an optimal aggregator based on that model as a proof of concept. Finally our *"experimental"* approach, split into a theoretical complexity analysis and performance comparison, is used to pinpoint all performance benefits and cost amortization gained with our approach.
- C3: The definition of iterative processing primitives is the result of a *"literature review"* of the state of the art and relevant large scale deployments. Furthermore, structured loops and the underlying decentralized progress tracking protocol combine the *"model"* method to define the dataflow graph embeddings combined with *"formal"* methods for proving the protocol's correctness and termination. Finally, we employ a *"build"* methodology related to the implementation of the multi-pass window aggregation and vertex-centric computational frameworks on top of structured loops.

1.6 Dissertation Outline

The rest of the dissertation goes as follows: chapter 2 covers the basic programming model and runtime characteristics of interest in Apache Flink as well as offering an overview of the topics presented in this work within that scope. In chapter 3 we present the problem of reliable stream processing for stateful streaming and the adoption of asynchronous epoch commits to solve major state management needs. Within chapter 4 we exhibit usages and cover issues related to the implementation of asynchronous epoch commits in Apache Flink. Then, chapter 5 covers the semantics and aggregation strategies for sliding windows while offering a new, optimised approach to declaring and executing stream windows. In chapter 6 we further analyze the problem of stream iterations and our complete design to tackle it further. Finally, we summarize our findings and discuss future work in chapter 7

CHAPTER
2

Background

This chapter gives a general description the Apache Flink system which served as a vehicle in this work. First, we cover Flink's programming model and the compilation of user programs to distributed long-running tasks. Then, we give an intuition of the problems at hand and their requirements in respect to Flink's semantics and execution.

2.1 The Apache Flink Platform

Apache Flink has been an open-source, top-level Apache project since January 2015. The overall system in its current form (release 1.5.0) is the result of the collective efforts of its growing community, including contributions of the author of the current dissertation. The purpose of this section is to provide a general overview of key elements in Flink's programming and execution models which have served as the common ground on which the contributions of this work build on (covered thoroughly within Chapters 3-6).

2.1.1 Overview of the Apache Flink Stack

Flink [46] provides a stack for programming, compiling and running distributed continuous applications. Programs in Flink are lazily evaluated upon submission to the runtime for execution. Essentially a Flink program is a declarative "prescription" of a distributed application and typically consists of a series of data stream transformations. In turn, executed programs are reconfigurable, distributed graphs of long-running executable tasks that are scheduled and monitored continuously by Flink's runtime components. In comparison to other known compute frameworks [8, 27] that are driven by short-lived, scheduler-driven execution, Flink dedicates compute cluster resources to applications as long as those applications are executed,

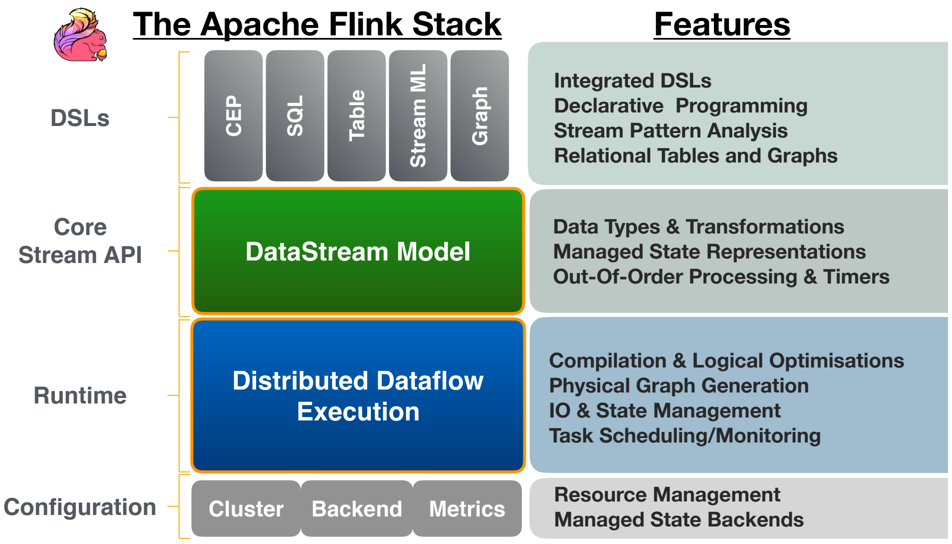


Figure 2.1: A breakdown of Apache Flink's software stack.

also known as "long-running task execution". In Figure 2.1 we break down the software stack of Flink from its high-domain specific libraries to runtime execution. In the sequel we give an overview of Apache Flink and further focus on the components of interest in this dissertation.

2.1.1.1 Programming Interface

Flink provides support for stream-based programming as a "shallow embbedded" Domain Specific Language (DSEL) through it's API provided in Scala and Java. Its programming interface consists of a core API and a number of specialized libraries built on top that provide special-purpose programming abstractions such as SQL-support. The core Stream API provides the basic primitives to build event-based applications such as managed application state and timers as well as end-to-end application building with stream transformations through using the `DataStream` API. Flink's DSLs include Gelly for graph processing, FlinkML for batch training models and on-line model serving, Table and Stream SQL for standing relational queries with table representations as well as CEP a declarative DSL for expressing stateful complex event processing and pattern matching on event streams.

2.1.1.2 Runtime

Flink's runtime is a distributed system that schedules and maintains applications and their corresponding tasks. As mentioned above, Flink employs a long-running task architecture. This means that tasks are being allocated to available slots and run permanently on dedicated threads, until the application shuts down manually or when reconfiguration is needed (see chapter 4). There are three types of runtime components that manage a Flink cluster: the Client, JobManager and TaskManagers. The Client module compiles and optimizes each user program into a logical graph representation that can be submitted to the JobManager for execution. The JobManager is the *master* process in a Flink deployment and has full control of an application and is responsible for the physical translation, deployment and monitoring of each logical graph submitted for execution. Finally, the TaskManager is a process deployed per-host which that carries out all local resource management (e.g., network/disk IO, memory buffers, slot allocation, configuration) needs for locally running tasks. Finally, Flink's runtime periodically commits the progress of the system to external stable storage and other systems using a transactional atomic commit protocol that executes concurrently with each distributed application (see chapter 3 and 4).

2.1.1.3 Setup Configuration

Flink allows for certain external modules to be configured upon demand, typically per Flink deployment or per-application through its configuration files. These include, most importantly, *state backends* which are subsystems specialized to materialize state operations such (e.g., Rocksdb [47]). Other modules are dedicated for metric collection and cluster management (e.g., YARN and Mesos [48, 49]) among others. In this section we focus on Flink's programming semantics, compilation and translation to task execution graphs. In chapter 4 we give a detailed description of certain runtime components and mechanisms of interest such as the Epoch Commit Protocol that drives Flink state management.

2.1.2 Programming Model Overview

Stream Applications in Apache Flink resemble the fluid, functional programming syntax of DryadLinq [50] and Apache Spark [27]. The Core API of Flink provides a set of basic abstract types such as `DataStream` and `WindowedStream` each of which supports transformations that are higher-order functions such as `filter`, `map`, `flatmap` and `reduce`. In principle, each transformation represents a logical graph operator that has data dependencies to other operators in a conceptual (dataflow) graph. We will demonstrate the core capabilities of Flink's programming model as well as the path from compilation to distributed execution by using an example.

Example: Consider a use-case where a set of different sensors periodically generate and submit to a partitioned log (e.g., Kafka queue [51, 52]) their current temperature. Assume that we would like to build a reliable service that reads these events and does the following: 1) computes every 8 min the average temperature measured by each sensor over the course of 1 hour and 2) reports warnings if an average temperature exceeds 40 degrees, while 3) ignoring the initial 5 measurements per sensor during its calibration period.

Listing 2.1: A Flink Scala Application for Analyzing Sensor Temperatures

```
1   case class SensorEvent(sensorID: Long, temperature: Double);
2   case class TemperatureWarning(sensorID: Long, temperature: Double);
3
4   //STREAM SOURCE
5   val sensors : DataStream[SensorEvent] = env.addSource(KafkaConsumer("sensorTemperatures"));
6   //--------------------------------------------------------------
7   //STREAM TRANSFORMATIONS
8   val filteredEvents :DataStream[SensorEvent] =
9     //Distribute Computation by SensorID
10    sensors.keyBy{_.sensorID}
11    //Ignore the first 5 temperatures per sensor
12    .filterWithState((evt:SensorEvent, count: Option[Int]) =>
13        count match{
14          case Some(c) => (c > 5, Some(c+1))
15          case None => (true, Some(1))
16        });
17  val avgTempStream: DataStream[SensorEvent] =
18    //Compute 1h rolling average of temperature/sensor every 8min
19    filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
20    .aggregate(AverageTemperature());
21    //Output warnings for average temperatures above 40 degrees
22  val warnings: DataStream[TemperatureWarning] =
23    avgTempStream.flatMap((evt, collector) => if(evt.temperature > 40)
24      collector.collect(TemperatureWarning(evt.sensorID, evt.temperature)));
25  //--------------------------------------------------------------
26  //STREAM SINK
27  warnings.addSink(KafkaProducer("tempWarnings"));
28
29  //EXECUTION
30  env.execute()
```

In Listing 2.1 we present a Flink program that implements the aforementioned sensor event analysis service using various core programming styles and features provided in the framework. A Flink program begins with a declaration of one or more data sources. A data source can be a bounded data resource such as a text file or an unbounded one such as an active message log. The Flink framework provides already a set of integrated connectors that can be used out of the box, in this case we use a Kafka connector that consumes messages from a message queue identified as "sensorTemperatures". The core logic of a Flink application consists of a set of pipelined transformations.

The `keyBy` operation (line 10) declares the key used to partition computation and state. Despite the fact that its semantics look similar to a relational `groupBy`, the `keyBy` does not group events but rather defines how events will be shuffled to different operator instances (events of the same key are processed by the same instance) and how state is instatiated, using that key. The effect of `keyBy` becomes more clear in the `filterWithState` transformation (line 12) which is parametrized with a user-defined function of type `(event,state)` $\rightarrow$ `(bool,newState)` that yields *true* for events to keep in a stream alongside a new state (instantiated per

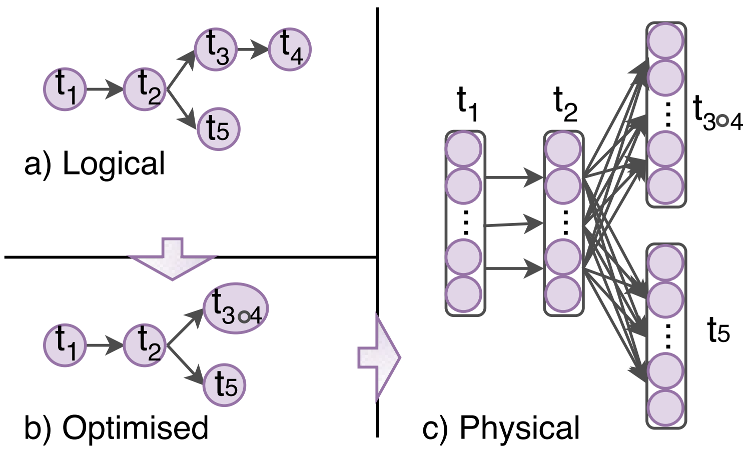


Figure 2.2: Translation from Logical to Physical Execution Graphs.

event key). The declared function maintains a count of events per sensor as a state and filters out events with a count less than 5 (line 14). During the runtime of the program Flink automatically manages the persistent state of each key (sensor id) which in this case is just an integer value (complex types are also allowed).

The partitioned filtered events are then fed into window aggregate transformation (lines 19 and 20). The `timeWindow` transformation discretizes the event stream into sets based on the application times that those are generated and applies an average function on these sets to derive a `SensorEvent` object of each sensor per window carrying the average temperature value. After that, the `flatMap` function (line 24) generates a `TemperatureWarning` event for each aggregate that has a temperature beyond the threshold (40 degress). Notice that in the *flatMap* function that allows mapping each input event into zero or more output events. In this case an output collector is passed to the user-defined function which represents an output buffer used to selectively flush output events in an imperative control flow logic (within functional programming literals). Finally, a Kafka stream sink is defined (line 27) to commit the output of the computation, i.e., the generated warnings.

2.1.3 Compilation and Execution Overview

Each program in Flink goes under a three-phase compilation steps before it gets executed in the distributed runtime that produce a 1) logical, 2) optimised and 3) physical representation graph respectively (Figure 2.2). In the previous example we examined the general composition of a Flink program.

Logical Representation: Each transformation in a program resembles a logical operator that executes event-based logic. Once a program reaches its execution command (line 30) the Flink runtime constructs at the client side a graph representation of all logical operators in the Program. The logical representation (Figure 2.2 (a)) $G^L = \{\Pi^L, \mathbb{E}^L\}$ is a directed acyclic graph whose source and sink nodes represent the ones declared in the program. An operator $p \in \Pi^L$ can encapsulate the logic of a single transformation (e.g., `filter`, `flatmap`, `window`). However, standard logical dataflow optimisations such as fusion [53, 54] are also applied in an intermediate step allowing multiple operators to be coalesced into one (Figure 2.2(b)).

Physical Representation: The mapping of a logical graph G^L to $G = \{\Pi, \mathbb{E}\}$, the physical, distributed execution graph (Figure 2.2(c)) occurs when a pipeline is deployed, on its initial run or upon reconfiguration (e.g., for scale-out). During that stage each logical operator $p \in \Pi^L$ is mapped to a number of physical units that we call "tasks" $p^1, p^2, \ldots, p^\pi \in \Pi$, each of which gets deployed to available containers throughout a cluster (e.g., using YARN [48] or Mesos [49]) according to a decided (or inferred/available) degree of parallelism $\pi \in \mathbb{N}^+$. The data dependencies between deployed tasks in Π are represented by configured channels that allow tasks to send and receive messages during their execution (e.g. using a `collector` in the operator logic).

2.1.3.1 Example of an Execution

In Figure 2.3 we depict each phase of the example program shown previously in Listing 2.1 starting from its logical representation up to each physical deployment and execution. In the optimised graph the `window, aggregate` and `flatMap` operators are coalesced into a single optimised operator (Figure 2.3(b)). Collocating the aggregation logic of a window with its discretization is generally necessary to avoid unnecessary and redundant computation (chapter 5 covers this topic thoroughly). The `flatMap` operator is also included in the same coalesced operator since no `keyBy` or other parallel data subscription is defined in between.

The final physical graph (Figure 2.3(c)) consists of multiple instances of each operator, each of which we typically call a task. Each task is invoked upon an input message in one of its channels, runs a local computation that can access its state for reads and writes and then generates outputs messages in a single processing action. Examples of state from our example application are the counts kept within the `filterWithState` operator or the state of each `window` (grouped records and their aggregates). The Kafka producers and consumers also keep state internally (e.g., log offsets) that is necessary for Flink's transactional commit protocol (chapter 4). State is typically exposed per-key, however each task is statically allocated with a group of keys for which it is responsible for. That means that 1) all records of a respective key are being routed by the system's runtime to that respective physical task and 2)

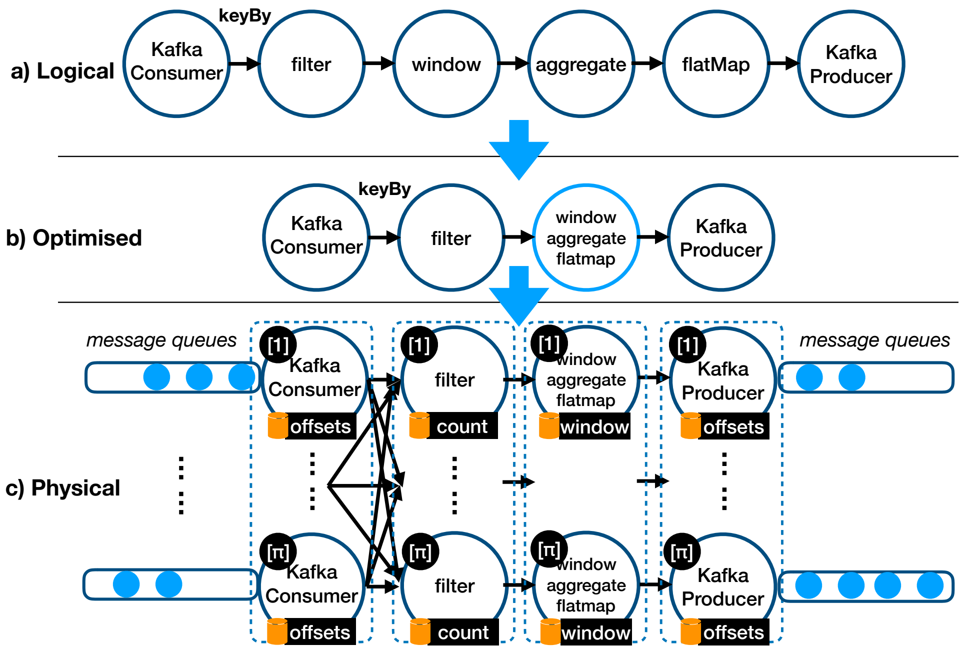


Figure 2.3: Logical, Optimised and Physical Representations of example program.

state entries of all keys are co-partitioned in the same physical partition that the responsible task resides. The exact specification of the fundamental execution model employed in each task is further analyzed in detail in chapter 3 of this dissertation alongside the system-wide transactional processing model. Furthermore, runtime concerns particular to Flink such as the key collocation strategy (key-groups) and the exact transactional state commit protocol (asynchronous epoch commit) are further analyzed in chapter 4.

Note: Throughout this dissertation we will use the term “task” to refer to physical (execution) tasks and the term “operator” to refer to either the logical operator or the *logic* that is being executed within a task. For example, a window operator gets deployed into tasks that execute the window operator logic (or simply a window operator) respective to each task.

2.2 System Challenges of Interest

Having covered the core concepts behind Flink, in this section we will address some major system design needs in Apache Flink and beyond which served as the main intuition of the overall work presented in the dissertation.

2.2.1 Managed Application State and Reliability

As of the time of writing (Flink v.1.5) each stream transformation in Flink can declare state that is persistent and always-available for read and write operations. State is a main building block of a program as it encapsulates, at any time, the context of each computation.

State Types: There are conceptually two scopes upon which managed state operates. For purely data-parallel stream operations such as for example a per-key average, the computation, its state and associated streams can be logically scoped and operate independently for each key (i.e., when `keyBy` is employed). We refer to this state as `Keyed-State`. For local, per-task computation such as for example a partitioned machine learning training model or offset-based partitioned log consumption (e.g., Kafka data sources), state can also be declared in the level of a physical task, known as `Operator-State`. Both `Keyed-State` and `Operator-State` are transparently partitioned and managed by the runtime of the system and are in most cases locally available to each task.

Embedded State: The choice of embedded state has been one of the very initial design decisions made during the inception of the system. This particular design choice had been proven to work well in prior work [10, 11] due to the avoidance of remote state access invocation, however it also introduced challenges. Initially, the most critical issue in Flink had been the need for fault tolerance. Flink programs are intended to run continuously for arbitrary periods of time, from minutes to years and thus the probability of a machine failing during the lifetime of a program is extremely high. Having a set of uncoordinated, independently running tasks and no global unit or "step" in a computation such as the highly attractive RDD model of Spark [12] made operations with Flink (during its prototype phase) hard and uncertain to reason. On top of local state, committing end-to-end side effects (e.g., output messages from sinks) reliably had also been an open problem for years. Flink's biggest users also demanded support for scale-out in their programs, being able to reconfigure their jobs on the fly and change parallelism or simply migrate tasks to new containers.

Contributions: The overall work on state management aimed to offer a universal approach to all these state management issues, complementing with Flink's original fine-grained task execution model while not sacrificing the correctness of programs. In chapter 3, we offer a formal system specification and description of the underlying "Asynchronous Epoch-Based Commit" execution model that is based on the ability to capture the complete configuration (i.e., task states) of a distributed application without halting compute tasks or enforcing any means of blocking synchronization. While existing protocols implement snapshots asynchronously (e.g., Chandy and Lamport's algorithm [28]) no prior solution has tackled the problem of capturing distributed states on complete predefined computational frontiers before, a more restricted form of a snapshot. In chapter 3 we also describe how to deal with cycles in a physical stream processing graph and further identify the necessary safety and liveness properties of epoch-based snapshotting and prove their satisfaction. Finally, in chapter 4 we summarize all implementation details and different applications of the snapshotting mechanism as well as discussing performance metrics from long-running production deployments of Flink.

2.2.2 Window Computation Sharing

In Flink and other stream processors [55, 42, 39], stream windows serve as the sole way to group continuous data into evolving sets and execute computation on those sets. In the previous example of Listing 2.1 we demonstrated briefly the usage of built-in application-time periodic windows. Periodic windows are typically declared using a *range* and a *slide*. In the same example, these parameters were 1 hour and 8 minutes respectively. A more careful observation of that particular operation can reveal an embarrassingly redundant amount of underlying computations involved across overlapping windows. For example, if we consider an average rate of 1000 records per second in application-time it would mean that 52 out of 60 minutes per window correspond to computation that has been recomputed in the past, circa 52000 aggregations. In fact, this is only the tip of the iceberg when it comes to the underlying needs for window computation sharing which we can summarize sorted by complexity (increasing) as follows:

1. Partial Window Aggregation: This problem is limited to how we evaluate each individual window. There are two strategies: a) Keep all records of a window until it is complete and then trigger the computation on the whole window, or b) Use a partial aggregate that is updated incrementally per input record.

Listing 2.2: Non-Incremental Window Average Example

```
val avgTempStream: DataStream[SensorEvent] =
  filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
  .process(AverageTemperature());

class AverageTemperature extends ProcessWindowFunction[SensorEvent, SensorEvent, Long,
      TimeWindow] {
  def process(key: Long, context: Context, input: Iterable[SensorEvent], out:
        Collector[SensorEvent]): () = {
    var sum = 0.0;
    var count = 0L;
    for (evt <- input) {
      sum = sum + evt.temperature;
      count = count + 1;
    }
    out.collect(SensorEvent(key, sum / count))
  }
}
```

Windows are bounded sets but can potentially be arbitrarily large (e.g., each containing billions of records), thus, strategy (b) is generally preferred since it eliminates the purpose of maintaining arbitrary window state and guarantees constant space requirements. As covered in chapter 5, this is possible when the aggregation function is distributive and associative. Flink's programming model provides support for both strategies using the *ProcessFunction* and *AggregationFunction* respectively. In Listings 2.2 and 2.3 we show an example of both alternatives respectively for the same window average of sensor temperatures introduced previously in Listing 2.1. The incremental version allows the user to break down aggregation into a partial aggregate type (in this case *(Double, Long)* corresponding to a sum and a count) and a final result extracted from that type (the average value).

Listing 2.3: Incremental Window Average Example

```
val avgTempStream: DataStream[SensorEvent] =
  filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
  .aggregate(AverageTemperature(), ResultEvent());

class AverageTemperature extends AggregateFunction[SensorEvent, (Double, Long), Double] {
  override def createAccumulator() = (0.0, 0);

  override def add(evt: SensorEvent, partial: (Double, Long)) =
    (partial._1 + evt.temperature, partial._2 + 1L);

  override def getResult(partial: (Double, Long)) = partial._1 / partial._2;

  override def merge(a: (Double, Long), b: (Double, Long)) =
    (a._1 + b._1, a._2 + b._2);
}

class ResultEvent extends ProcessWindowFunction[Double, SensorEvent, Long, TimeWindow] {
  def process(key: Long, context: Context, averages: Iterable[Double], out:
        Collector[SensorEvent]): () = {
    val average = averages.iterator.next();
    out.collect(SensorEvent(key, average));
  }
}
```

2. Periodic Window Aggregation-Sharing: We have seen how partial aggregation optimizes computation per window, however, this approach does not solve the general problem presented earlier, as is. This is due to the fact that even partial, incremental computation *across* sliding windows would still be duplicated multiple times, one per particular window. In the previous example of range 60min and slide 8min, around 8 windows would be active at any time (perhaps even more if windows are evaluated out-of-order). To deal with this issue, most existing systems employ a class of implicit computation sharing techniques called "slicing" [13, 14] which effectively pre-computes non-overlapping partials on top of stream segments known a priori at compilation time (since windows are periodic).

3. Periodic Multi-Window Aggregation Sharing: It is quite common for an application to contain aggregates over multiple window specifications (e.g., when building data warehousing measures or analyzing simulated scientific models). For example, in Listing 2.4 we have three window averages specified on different yet highly overlapping time windows.

Listing 2.4: Periodic Multi-Window Average Example

```

val avgTempStream1: DataStream[SensorEvent] =
  filteredEvents.timeWindow(Time.hours(1), Time.minutes(8))
  .aggregate(AverageTemperature(), ResultEvent());

val avgTempStream2: DataStream[SensorEvent] =
  filteredEvents.timeWindow(Time.hours(2), Time.minutes(30))
  .aggregate(AverageTemperature(), ResultEvent());


val avgTempStream3: DataStream[SensorEvent] =
  filteredEvents.timeWindow(Time.minutes(20), Time.minutes(5))
  .aggregate(AverageTemperature(), ResultEvent());

...
```

This is an interesting optimization problem that has been also tackled effectively in the past mainly for periodic windows with composite slicing optimization schemes decided at compilation time [14]. Yet, when it comes to non-periodic windows existing solutions fall back to eager pre-aggregation data structures [32, 15] that incur high memory footprint.

4. Any-Window Aggregation Sharing: Nowadays windowing semantics are becoming increasingly complex. In addition to time-based, systems like Flink, Beam or Apex [46, 42, 39] allow for even more built-in window specifications such as session windows [21]. Furthermore, apart from Flink and Beam, systems such as IBM Streams and its SPL language [56, 57] offer the possibility for users to write assigners for user-defined windowing logic or custom policies that specify how records can be bucketed into windows purely based on the runtime state of the application. Windows are also often defined based on the data trends that can change over time [58]. The bottom-line here is that most complex windows cannot be known apriori and therefore it is hard for a system to choose the right optimization strategy for computation sharing on either single or multi-window specifications.

Contributions: As described thoroughly in chapter 5, we offer a universal solution to all aforementioned problems. Our approach is twofold: we first introduce the notion of user-defined deterministic windows that can be used to slice (create shared partial aggregates) at-runtime for a plethora of window specifications. We then share all runtime-specific slices in a pre-aggregation data structure that can be used to derive seamlessly any full window aggregate.

2.2.3 Multi-pass Window Aggregation

An acyclic control flow in a program is not always enough to solve a computational problem. The same principle known from imperative programming applies to most programming paradigms including data stream processing. Stream processing frameworks such as the one provided currently by Flink allow complex operations (e.g., sliding window aggregations) and synchronization primitives (e.g., global timers on event-time) on streams, yet they fall short to integrate cyclic computation both in terms of programming semantics and runtime support. In our study, we begin to seek the right cyclic execution model while having Flink's existing architecture as a baseline.

Out-of-Order Processing: Stream windows are an attractive starting point to employ iterative processes on top. That is mainly due to the existing build-in support for parallel execution and determinism, especially in the context of event-time windowing. Event time cross-relates events with a total order relation, that of their time of generation at their respective source, mapping each event to a particular *earth time*. Evidently, since local clocks are not in sync with a global atomic clock and real-world networks stand between (and within) systems events can be processed out of order [23]. Flink and most other stream processing systems nowadays implement an intermediate model for tracking time progress called out-of-order processing (OOP) model [59], which is based on the monotonic progression of time up to a predefined bounded degree of un-orderness. This allows for substituting local process clocks with *progress metrics*, which are monotonic measures used to implement event time-based synchronization barriers across parallel tasks (e.g. event-time windows). Progress metrics have been in wide use within production-grade systems and are typically implemented via the use of low watermarking[9, 22, 59], a decentralized technique that works correctly on the embedded state model. The applications of progress metrics and OOP are mainly centered on defining when certain computation can be triggered rather than how it is optimized or shared and thus, OOP was of no interest during our initial research work in window aggregation sharing (chapter 5). However, it has been really influential for our work in iterative stream processing.

Contributions: In chapter 6 we provide a formal model specification and complete integration of iterative progress metrics to Flink's existing stream process model specification (chapter 3). Semantically, our approach borrows ideas from the timing model of Naiad [20] a high performance processing system which introduced the notion of cyclic progress tracking. Yet, rather than having a separate coordination mechanism for cyclic computation, we rather fuse the required additional primitives to the existing production-proof and purely decentralized low-watermarking mechanism. As a showcase, we extend Flink's existing single-pass window aggregation model to multi-pass aggregation with tunable synchronization staleness.

CHAPTER

3

Reliable Data Stream Processing

3.1 Introduction

Scalable data stream processing platforms such as Apache Flink are distributed systems. A distributed system consists of multiple processes connected through a network, that send and receive messages. Distributed systems are typically designed to make all concerns related to their distributed nature transparent to the user, offering the view of a single entity. The same transparency principle simplifies not only the programming model but also the work of a system designer. In this chapter, we address a crucial design challenge when it comes to distributed stream processing, that of guaranteeing a reliable unbounded execution. Reliable processing relates to guarantees offered by a continuously running distributed system despite partial failures or reconfiguration phases that can occur.

In this study, we first revisit the concept of consistent snapshots, which are replicas of the global configuration (i.e., state) of a system at a specific point of its distributed execution. Consistent snapshots capture a complete distributed state that can be used as a single atomic reference, to inspect [28], recover [60] or even alter a distributed computation. Most existing snapshotting protocols are distributed algorithms that make certain assumptions regarding the structure of a system (e.g., strongly connected process graph) to collect all process and channel (in-transit messages) states which are part of a snapshot.

A variety of operational challenges in distributed stream processing relate to guaranteeing that every process in the system consumes its input messages, updates its internal state and generates output messages without loss, even while independent process failures occur or the configuration of the system changes [11, 9, 61]. Past approaches [11, 62, 63] aimed to tackle this challenge via fine-grained communication protocols between individual processes to re-conciliate lost state (e.g., replaying input), filter out duplicates or restricting semantics to idempotent operations [9]. While some of these techniques solve individual special cases of reliable processing, they do not offer a clear specification of the guarantees they implement. More importantly, they miss one of the most fundamental needs in stream processing, that of a unified state management mechanism. Unified state management should cover all aspects of application state, such as migrating, reconfiguring, querying, recovering and versioning the unbounded execution of a stream processing application.

In this chapter, we propose an execution model for stateful stream processing based on the notion of epochs. Epochs define concrete points in a distributed stream execution where state is atomically committed and thus, they can be used to provide unified state management. We argue that epochs do not enforce a discretized execution [12] and can instead be committed asynchronously using a special form of distributed snapshotting aligned on epochs. Starting from Chandy and Lamport's original algorithm, we present all necessary modifications needed to respect epoch order and prove the respective provided properties. Our final distributed protocol [30, 29] manages to capture the complete state of a weakly connected stream process graph [20, 64, 9, 54, 11] after an epoch, limiting channel state logging (in-transit messages) to graph cycles that optionally exist. Finally, this chapter serves as a self-contained specification of the epoch-based stream processing, its asynchronous implementation with snapshotting and its foundations. Chapter 4 presents a concrete usage and implementation of epoch-based snapshots for various operational needs in Apache Flink.

The chapter's outline goes as follows: section 3.2 offers an overview of consistent snapshotting in the fail-stop process model and related safety and liveness properties. Our preliminary analysis covers the concept of marker-based snapshotting, having Chandy and Lamport's protocol and its assumptions as a starting point, followed by several direct yet important generalizations. In section 3.3 we model the execution of a data stream processing system as a special case of interest and discuss the problem of reliable processing guarantees in that model as well as shortcomings of the related state of the art. Section 3.4 presents the specification and protocol for epoch-based stream processing, our proposed solution to reliable data stream processing. Finally, section 3.5 offers a summary of our approach.

3.2 Preliminaries

In this section we present a complete model and specification for consistent snapshots, identifying the necessary properties and known distributed algorithms that solve this problem on any message-passing system. Then, in section 3.3, we look into a more restricted version of snapshots, relevant to the problem of epoch-based stream processing.

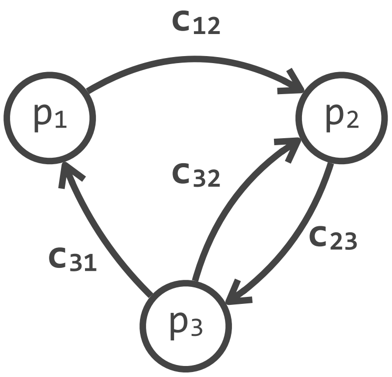


Figure 3.1: A process graph with three processes.

3.2.1 System Model

A distributed system model [60, 28] consists of two core components: processes and network channels. We assume a finite set of processes Π as well as a finite set of network channels $\mathbb{E} \subseteq \Pi \times \Pi$. For any two processes $p_i, p_j \in \Pi$ p_i can send messages to p_j iff $(p_i, p_j) \in \mathbb{E}$. In principle, we can represent a distributed system as a directed graph $G = (\Pi, \mathbb{E})$ with Π being its vertices and $\mathbb{E}$ representing its edges. For brevity, we will denote a channel $(p_i, p_j) \in \mathbb{E}$ simply as c_{ij}. In Figure 3.1 we depict the process graph $G = \{\{p_1, p_2, p_3\}, \{c_{12}, c_{23}, c_{31}, c_{32}\}\}$ as an example.

Process Model: Each process $p \in \Pi$ has its own local volatile state s_p that is initially empty. Furthermore, it is statically initialized with a number of input channels $\mathbb{I}_p = \cup_{x \in \Pi}\{c_{xp} | (x, p) \in \mathbb{E}\}$ and output channels $\mathbb{O}_p = \cup_{x \in \Pi}\{c_{px} | (p, x) \in \mathbb{E}\}$. In certain cases we will identify the channel on which a message is sent or received (e.g., m_{qp} or $m_{c \in \mathbb{O}_p}$), when that is necessary.

Channels and State: Channels are communication endpoints, implementing certain behavior (e.g., FIFO delivery) between processes and we decouple them from the in-channel state M, a multiset of all in-transit messages in the system (i.e., all messages sent but not yet delivered). Each channel $c_{pq} \in \mathbb{E}$ supports two types of events: $\langle send, m \rangle$ and $\langle recv, m \rangle$. Sending a message m results into $M \rightarrow M \cup \{m\}$ while receiving that message results into $M \rightarrow M/\{m\}$. For simplicity, in this context we assume that point to point network channels are reliable and have unbounded capacity, thus, they cannot overflow or lose messages that are in transit. Specification 2, presented later in the chapter, provides a formal description of a channel interface and properties for FIFO reliable channels.

Execution Model: The complete state or *configuration* of the system can be summarized by $\{\Pi_*, M\}$, where $\Pi_* = \{s_p | \forall p \in \Pi\}$ and M its in-transit messages. An execution of a system can be modeled via the use of transitions to its configuration $\{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\}$, where each transition is caused by a primitive step or *action* executed by a single process $p \in \Pi$ and is summarized as (s_p, m, M_{OUT}, s'_p) as follows:

- s_p: The local state of p before the action.
- m: An input message $m \in M \cup \{\varnothing\}$.
- M_{OUT}: A set of output messages M_{OUT}.
- s'_p: The local state of p after the action.

For a process p, an action consists of individual events that respect the following event flow: 1) The reception of an input message m that causes that action (if $m = \{\varnothing\}$ then it is an *internal* action), 2) An internal computation: $(m, s_p) \rightarrow (s'_p, M_{OUT})$ and 3) a set of *send* events that put messages M_{OUT} into outgoing channels. After an action on $\{\Pi_*, M\}$ the new system configuration $\{\Pi'_*, M'\}$ contains the updated local process state $\Pi'_* = \Pi_* \cup \{s'_p\}/\{s_p\}$ and a new set of in-transit messages $M' = (M \cup M_{OUT})/\{m_{IN}\}$. For simplicity, in the most part we will refer to single-event actions such as $\langle send, m_{qp}\rangle_q$ and $\langle recv, m_{qp}\rangle_p$, each of which results into a new process state, unless stated otherwise.

Local and Global Execution: Sequences of events in a system form an execution E. An execution contains all events that have occured from the initial state of the system up to the events of the latest action. We often differentiate between E_p, the subset of an execution that contains events occured in process p and the global execution $E = \cup_{p \in \Pi} E_p$ that contains all individual events occurred across all processes in a distributed system.

Local Event Order: For each respective process $p \in \Pi$ computation follows a natural sequence of state transitions from an empty or initial state s_p^0 as such: $s_p^0, s_p^1, \ldots, s_p^n$. The respective events that cause these transitions $e_p^0, e_p^1, \ldots, e_p^i$ are totally ordered by a local causal order relation $\leq_p$, instrumented by its strict underlying local execution $E_p = \{e_p^0, e_p^1, \ldots, e_p^i, \ldots\}$ such that $e_p^i \leq_p e_p^j$ iff $i < j$. Since local event order is a total order it satisfies antisymmetry, totality and transitivity.

Failure Model: We assume a fail-stop model according to which a process stops and loses its current volatile state when a fault occurs. We will further refer to a process as *correct*, always in respect to an *observed* execution E when no fault occurs within E. We further denote a set of correct processes $\Lambda \subseteq \Pi$. The observed execution is context specific and contains all events that occur within the execution of a protocol (e.g., the consistent snapshotting protocol in subsection 3.2.3).

3.2.2 Consistent Cuts and Rollback Recovery

The concept of rollback recovery has many usages in distributed computing. It is, in essence, a reconfiguration process that aims to restore the execution of a full distributed system back to an intermediate point that was captured during a failure-free execution. A full system rollback can often be used as a mechanism for fault recovery. This is especially important in long-running system executions in order to avoid a complete re-execution of a computation that could take days or months. The main prerequisite of rollback recovery is to first be able to capture and copy the global state (configuration) of a system to stable storage, i.e., all process states as well as messages that are being in-transit.

Evidently, in a fail-stop model with volatile local state it is impossible to ensure that a complete system configuration will be recorded at the same instant due to the absence of a global atomic clock [28]. What is otherwise achievable by global state snapshotting techniques is to capture a "valid" configuration, one that can possibly occur during any failure-free execution [65]. A more acceptable formal interpretation of which global state can be considered valid lies on the definitions of causal event order and consistent cuts.

3.2.2.1 Global Causal Order

Given a global execution $E = \cup_{p \in \Pi} E_p$ including all individual events occurred across all processes in a distributed system, it is impossible to derive a total order relation for all events happening concurrently without a global atomic clock. However, there exist a partial order relation, known as *causal order* [66], given in Definition 3.2.1.

Definition 3.2.1. Given an execution E and $e, e', e'' \in E$, the *causal partial order* relation $\prec$ satisfies the following:

(1) $e \leq_p e' \implies e \prec e'$.

(2) if e and e' correspond to $\langle send, m_{pq} \rangle_p$ and $\langle rcvd, m_{pq} \rangle_q$ then $e \prec e'$.

(3) $(e \prec e') \wedge (e' \prec e'') \implies e \prec e''$ *(Transitivity)*.

The transitivity of causal partial order is especially important for specifying the order relation between two events that have otherwise occurred across two different processes in the system. Event diagrams, as the one depicted in Figure 3.2 are often used to visually represent executions of events in a distributed system where dots on each process line show events in the local execution of each process and arrows pointing out the causal dependencies of respective send and receive events in that execution between processes. In Figure 3.2 we highlight the casual relation between event a and d, b, e, f, g, h all of which belong to a's transitive closure of $\prec$ along the

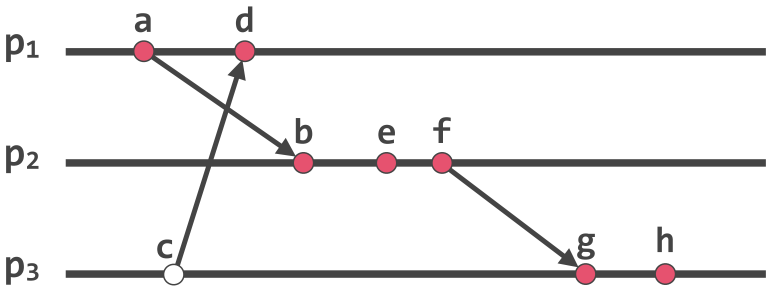


Figure 3.2: An event diagram highlighting events causally related to a.

causal relation path in the diagram. In the case of event c we cannot specify any causal order with respect to a since c is not within the transitive closure of a and vice versa, thus, in that case $a||c$ (i.e., a and c are concurrent).

3.2.2.2 Valid Configurations and Consistent Cuts

A concept that is strongly related to the notion of distributed snapshots is a *distributed cut*. Definition 3.2.2 covers the most general case of a distributed cut. Notice that here we are including only local events preceding e.

Definition 3.2.2. Given an execution E and $e, e' \in E$, a distributed cut is a set $\mathcal{C} \subseteq E$ which satisfies the following invariant: $(e \in \mathcal{C} \wedge e' \leq_p e) \implies e' \in \mathcal{C}$

Example: A cut can be visually represented as a line that "cuts" through an execution, marking a frontier where all *locally* preceding events are in the cut. Figure 3.3 depicts two such cuts based on the process graph of Figure 3.1. In the case of cut C_1 (Figure 3.3(a)), only the events $\langle send, m\rangle_{p1}$ and $\langle rcvd, m'\rangle_{p1}$ are included in the cut. While C_1 is a possible distributed cut it cannot represent a valid state of the system since according to the included events, message m' was received but never sent, which is impossible. There is simply no possible execution E that can lead to such configuration. On the other hand, cut C_2 (Figure 3.3(b)) represents a valid configuration in the same execution, one where every event received was previously sent (event m' was only sent according to C_2 which does not violate any of the properties).

A consistent cut, described in Definition 3.2.3, is one that respects causal partial ordering and therefore, captures the notion of a "valid" system configuration.

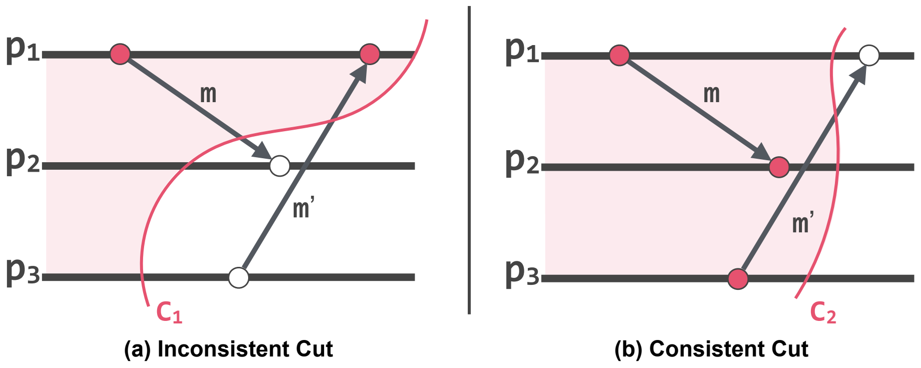


Figure 3.3: An example of an inconsistent (C_1) and a consistent cut (C_2).

Definition 3.2.3. Given an execution E and $e, e' \in E$, a *consistent distributed cut* is a set $\mathcal{C} \subseteq E$ which satisfies the following invariant: $(e \in \mathcal{C} \wedge e' \prec e) \implies e' \in \mathcal{C}$. Furthermore, a system configuration $\{\Pi_{\mathcal{C}}, M_{\mathcal{C}}\}$ after an execution bounded by a consistent cut $\mathcal{C}$ is also a *valid configuration*.

3.2.3 Consistent Snapshots: Specification and Fundamentals

Snapshotting protocols are distributed algorithms that capture a valid configuration of a distributed system during its execution. Since snapshots are distributed they have to be partially executed by each respective process in a coordinated manner which results into a consistent cut. Optimally, as described by Chandy and Lamport [28], the execution of a snapshotting protocol should not interfere with an application but run concurrently with it, capturing states when necessary while the application is running. In practice, a snapshot $S_{\mathcal{C}} = \{\Pi_{\mathcal{C}}, M_{\mathcal{C}}\}$ captures a valid configuration for *any* consistent cut $\mathcal{C}$. For brevity, we will imply that all snapshots are "consistent", unless stated otherwise.

Example: Consider a snapshot implementing the consistent cut C_2 that was presented previously in Figure 3.3(b). If we consider all events, messages and state transitions occurred, as depicted in Figure 3.4, it is more clear to visualize what the exact configuration of the system would be based on C_2. When it comes to process states, C_2 interleaves with processes p_1, p_2 and p_3 at states s_1^1, s_2^1 and s_3^1 respectively. Furthermore, message m' was in transit in the same cut (as being sent but never received). Thus, for completeness, message m' is also included in the snapshot. The full snapshot can be summarized as such: $S_{C_2} = \{\Pi_{C_2} = \{s_1^1, s_2^1, s_3^1\}, M_{C_2} = \{m'\}\}$.

Now let us define more formally the specification of consistent cuts, in terms of the interface of the protocol as well as the required safety and liveness properties that it should satisfy.

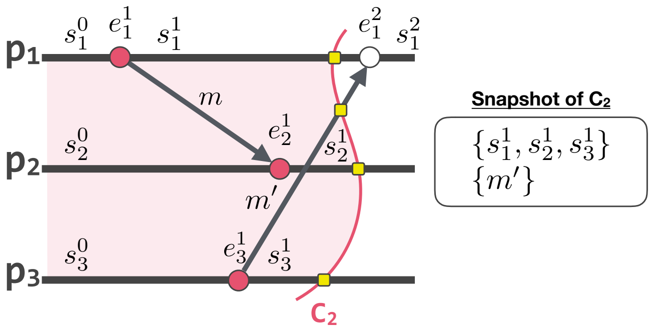


Figure 3.4: Contents of a snapshot for consistent cut C_2.

Specification 1: Consistent Snapshotting (csnap)

Event Interface:

Request: $\langle snapshot \rangle$: Initiates a consistent snapshot.

Indication: $\langle record|p, s_p^k, M_p \rangle : s_p^k \in \Pi_\mathcal{C} \wedge M_p \subseteq M_\mathcal{C}$.

Properties:

CSNAP1: *Termination*: $\Lambda = \Pi \implies \langle record|p, \_ \rangle \in E_p, \forall p \in \Pi$.

CSNAP2: *Validity*: Configuration $S_\mathcal{C} : \{\Pi_\mathcal{C}, M_\mathcal{C}\}$ is *valid*, where $\Pi_\mathcal{C} = \cup_{p \in \Pi} s_p$ and $M_\mathcal{C} = \cup_{p \in \Pi} M_p$.

Interface: Consistent snapshotting is more formally defined in Specification 1 (csnap) which summarizes the interface and expected safety and liveness properties to be satisfied by a system component that provides that functionality. The interface of the protocol contains two core messages: a request $\langle snapshot \rangle$ which initiates a consistent snapshot and $\langle record|p, s_p^k, M_p \rangle$ a response message that contains an internal state s_p^k as well as M_p a set of in-transit messages captured by process p.

Properties: We further break down the requirements of consistent snapshotting into two properties: *Termination* and *Validity*. Termination is a liveness property that is eventually satisfied if all processes are correct during an instance of a snapshot. The assumption that $\Lambda = \Pi$ might appear too strong, however, in the context of rollback recovery it is important to capture the complete application state. In case any failures occur, it is required to bring the whole system back to a global state captured during a failure-free execution, thus, no failures are tolerated during the execution of the protocol. The *Validity* property is a safety property that bears equivalence to the invariant of consistent cuts (see Definition 3.2.3). In essence, for every local event e that precedes the event e_p^k that caused state $s_p^k : e \leq_p e_p^k$ it is implied that $e \in \mathcal{C}$ (Lemma 3.2.1).

Lemma 3.2.1. For any process $p \in \Pi$ with snapshotted state $s_p^k \in \Pi_{\mathcal{C}}$ and event $e \in E_p : e \leq_p e_p^k \implies e \in \mathcal{C}$.

Proof. Lemma 3.2.1 derives from the specification of events as well as Definition 3.2.2. We know that a state s_p^k is caused through a state transition triggered by event e_p^k. Thus, $(1) s_p^k \in \Pi_{\mathcal{C}} \implies e_p^k \in \mathcal{C}$. From Definition 3.2.3 we also know that $(2)(e' \leq_p e'' \wedge e'' \in \mathcal{C}) \implies e' \in \mathcal{C}$. (1) and (2) satisfy Lemma 3.2.1. □

3.2.4 Snapshotting Strongly Connected Graphs

Several algorithms have been proposed for satisfying the specification properties of consistent snapshots, targeting process graphs with specific structural characteristics as well as network assumptions (e.g., tolerating message loss [67]) and initiation strategies. In respect to structural characteristics of a process graph, connectivity is more especially a critical property that needs to be taken into consideration in the design of a snapshotting algorithm. We refer to connectivity as the set of assumptions we can make about the reachability of every pair of processes $p, q \in \Pi$ as expressed in Definition 3.2.4.

Definition 3.2.4. Given a process graph $G = (\Pi, \mathbb{E})$ and processes $p, q, w \in \Pi$, the reachability relation $\sim$ satisfies the following:

(1) $\exists c_{pq} \in \mathbb{E} \implies p \sim q$.

(2) $p \sim q \wedge q \sim w \implies p \sim w$ *(Transitivity)*.

One of the most popular algorithms from classic computer systems literature is the one by Chandy and Lamport [28] (C-L for short). The C-L algorithm captures a valid system configuration and terminates if the protocol is initiated by any given process in a strongly connected graph. A strongly connected graph is a process graph for which every process is reachable by another process (Definition 3.2.5).

Definition 3.2.5. A graph $G : (\Pi, \mathbb{E})$ is *strongly connected* iff $p \sim q, \forall p, q \in \Pi$.

In the most general case, the C-L algorithm relies on a *strongly connected* process graph to guarantee termination. Furthermore, it depends on reliable network channels with strict FIFO delivery order for ensuring validity. Despite having relatively strict connectivity assumptions, the C-L approach has been highly influential to the current work since it exhibits the use of a non-coordinated and non-blocking mechanism for capturing the configuration of any strongly connected

Specification 2: FIFO Reliable Channel (`fiforc`)

Event Interface:

Request: $\langle send, m\rangle : M \rightarrow M \cup \{m\}$.

Indication: $\langle rcvd, m\rangle : M \rightarrow M/\{m\}$.

Properties:

FIFORC1: *Reliable Delivery*: $p, q \in \Lambda : m_{pq} \in M \implies \langle rcvd, m_{pq}\rangle \in E_q$.

FIFORC2: *No Creation*: $\langle rcvd, m\rangle \in E \implies m \in M$.

FIFORC3: *No Duplication*: $\forall \langle rcvd, m\rangle, \langle rcvd, m'\rangle \in E \implies m \neq m'$.

FIFORC4: *FIFO Delivery*: $\forall p \in \Pi, q \in \Lambda$
$\langle send, m\rangle_{pq} \leq_p \langle send, m'\rangle_{pq} \iff \langle rcvd, m\rangle_{pq} \leq_q \langle rcvd, m'\rangle_{pq}$.

component. In this section we describe the C-L, starting from the formal specification of FIFO Reliable Channels.

FIFO Reliable Channels: Specification 2 summarizes the interface and expected properties of a FIFO Reliable channel in the semantics of our model. For brevity, we assume that this component is instantiated per directional channel in the system (one endpoint at the sender and another at the recipient process) and thus omit to include the addresses of the processes in the events of its interface, whenever that is unnecessary. Typically, FIFO reliable channels can be implemented on top of reliable point to point links which in turn build on fair-loss links [68]. In practice, TCP channels adequately satisfy the properties of this abstraction.

Algorithm Semantics: Regarding the notation of the algorithms throughout this part of the dissertation, we adopt an event-based algorithmic specification [68] describing the logic executed by the protocol within message-handling function literals (**upon** `<msg>` **on** `<interface>`). We assume that message-handlers are executed by a single thread, thus, there are no concurrent executions of respective handlers when it comes to a single component instance. Events are triggered on interfaces (e.g., `fiforc` and `csnap`) and respect a provided specification. Furthermore, we use `<variable>` $\leftarrow$ `<value>` to mark assignments and `<interface>` $\rightarrow$ `<message>` for triggering messages on respective interfaces. In fact, every event handler implements an atomic action for that process (altering internal and network state). The internal computation (application logic) is executed within the `process:` $(m, s_p, \mathbb{O}_p) \rightarrow (s'_p)$ function and yields a new internal state and output messages (triggered on the channels passed).

3.2.4.1 Analysis of Chandy-Lamport Snapshots

In Algorithm 3 we present the C-L protocol by integrating its complementary message-handling logic within the regular operation of a process. The core intuition of the C-L protocol is to disseminate a special marker message "$\odot$" in the computational graph which acts as a seperator between those events that precede a consistent cut from those that come after the cut. This technique aids processes to decide with purely local information on when they can trigger a local copy of their state (at the instant they first receive a marker) as well as which messages should be part of the global snapshot (in case they causally precede a marker). In Specification 1 we describe the consistent snapshotting algorithm, as integrated logic in the process model.

3.2.4.2 Termination

The *Termination* property as formulated in Specification 1 indicates that the if all processes are correct and the protocol is initiated on any process of a strongly connected graph then the global recorded state should eventually be indicated along all processes in the graph: $\Lambda = \Pi \implies \langle record|p, \_\rangle \in E_p, \forall p \in \Pi$. Termination can follow a direct proof based on the deterministic order of local events, the reliable delivery of FIFO channels and the reachability properties of a strongly connected graph as follows:

Proof. Every process in the C-L algorithm indicates its recorded state once a marker has been received through all of its input channels. Thus, it trivially follows that every process indicates a full recorded state iff a marker is received through all existing channels in $\mathbb{E}$:

$$\langle record|p, \_\rangle \in E_p, \forall p \in \Pi \iff \langle rcvd, \odot\rangle_{\forall c \in \mathbb{E}} \in E \tag{3.1}$$

Thus, it suffices to prove that

$$\Lambda = \Pi \implies \langle rcvd, \odot\rangle_{\forall c \in \mathbb{E}} \in E \tag{3.2}$$

Now let us assume that the protocol is initiated at any process $p \in \Pi = \Lambda$. Via deterministic local ordering of events process p broadcasts a marker through its output channels that is eventually received through FIFO reliable channels:

$$\begin{aligned} \langle snapshot\rangle \in E_p &\implies \langle send, \odot\rangle_{\forall c \in \mathbb{O}_p} \in E_p \\ &\implies \cup_{q \in \Pi : c_{pq} \in \mathbb{O}_p}\{\langle rcvd, \odot\rangle \in E_q\} \text{ (FIFORC1)} \end{aligned} \tag{3.3}$$

Given the reachability of strongly connected graphs (Definition 3.2.4) we can provide via induction the following generalization:

Algorithm 3: Chandy-Lamport Consistent Snapshots

```
**Implements**: csnap, **Requires**: fiforc $(\mathbb{I}_p, \mathbb{O}_p)$
$(\mathbb{I}_p, \mathbb{O}_p) \leftarrow$ configured_channels;
$s_p \leftarrow \varnothing$;                                          ▷ volatile local state
$\mathsf{Recorded} \leftarrow \emptyset$;                              ▷ channels under logging
$s_p^* \leftarrow \emptyset$; $M_p \leftarrow \emptyset$;             ▷ state in snapshot

**Upon** $\langle \mathsf{rcvd}, m \rangle$ *on* $c_{qp} \notin \mathsf{Recorded}, m \neq \odot$
    $s_p \leftarrow \mathsf{process}(m, s_p, \mathbb{O}_p)$;           ▷ regular process logic
**Upon** $\langle \mathsf{rcvd}, m \rangle$ *on* $c_{qp} \in \mathsf{Recorded}, m \neq \odot$
    $M_p \leftarrow M_p \cup \{m\}$;                                   ▷ record in-transit message
    $s_p \leftarrow \mathsf{process}(m, s_p, \mathbb{O}_p)$ ;
**Upon** $\langle \mathsf{rcvd}, \odot \rangle$ *on* $c_{qp} \in \mathbb{I}_p$
    **if** $s_p^* = \mathsf{empty}$ **then**
        $\mathsf{startRecording}()$;
    Recorded = Recorded $-\{c_{qp}\}$;
    **if** Recorded $= \emptyset$ **then**
        $\mathsf{csnap} \longrightarrow \langle \mathsf{record}|\mathsf{self}, s_p^*, M_p \rangle$;
**Upon** $\langle \mathsf{snapshot} \rangle$ *on* $\mathsf{csnap}$
    $\mathsf{startRecording}()$;
    **if** Recorded = $\emptyset$ **then**
        $\mathsf{csnap} \longrightarrow \langle \mathsf{record}|\mathsf{self}, s_p, \emptyset \rangle$;
**Fun** *startRecording()*
    $s_p^* \leftarrow s_p$;                                           ▷ record local state
    **foreach** $\mathsf{out} \in \mathbb{O}_p$ **do**
        $\mathsf{out} \longrightarrow \langle \mathsf{send}, \odot \rangle$;
    Recorded $\leftarrow \mathbb{I}_p$
```

$$(3.3) \iff \ldots \iff \cup_{q \in \Pi : p \sim q}\{\langle \mathsf{send}, \odot \rangle \in E_q\} \tag{3.4}$$

and hence, via deterministic processing and FIFO reliable channels:

$$\begin{aligned}(3.4) &\iff \cup_{q \in \Pi : p \sim q}\{\langle \mathsf{rcvd}, \odot \rangle_{\forall c \in \mathbb{I}_q} \in E_q\} \\ &\iff \cup_{q \in \Pi : p \sim q}\{\langle \mathsf{send}, \odot \rangle_{\forall c \in \mathbb{O}_q} \in E_q\} \text{ (local event order)} \qquad (3.5) \\ &\iff \langle \mathsf{rcvd}, \odot \rangle_{\forall c \in \mathbb{E}} \in E \text{ (FIFORC1)}\end{aligned}$$

□

3.2.4.3 Validity

We should prove that every snapshot acquired via the C-L protocol is valid. It suffices to show that the events that are included in the cut compose a consistent cut in the execution of a system as it was shown in Theorem 3.2.1.

Proof. Given two events $e_p, e_q \in E$ for which $e_p \prec e_q$ we need to prove that if $e_q \leq_p e_q^k$ then this implies that $e_p \leq_p e_p^l$. From Lemma 3.2.1 we know that this condition satisfies the requirement of consistent cuts.

Let $e_p^\odot$ represent the local snapshot aquisition step within a process p. Based on the marker-forwarding logic of the C-L protocol, process p captures its local state s_p^k and forwards the marker further in the same computational step $e_p^\odot$, therefore $e_p^k = e_p^\odot$. No other local event can possibly occur between a $\langle rcvd, \odot \rangle$ and a respective $\langle send, \odot \rangle$ event to alter the state of process p. It therefore suffices to prove the following: for two events $e_p, e_q \in E$ for which $e_p \prec e_q$, if $e_q \leq_q e_q^\odot$ then this implies that $e_p \leq_p e_p^\odot$. We identify two possible cases (I) $(p = q)$ and (II) $(p \neq q)$.

(I) $p = q$

Given that $p = q$, let us call the two events e_p and e'_p, so that $e_p \leq_p e'_p$ (1). If e'_p is part of the snapshot, then $e'_p \leq_p e_p^\odot$ (2). From (1) and (2) it derives directly that $e_p \leq_p e'_p \leq_p e_p^\odot$ and therefore, e_p should also be part of the the cut.

(II) $p \neq q$

Assume that p and q are directly connected through a FIFO reliable channel $c_{pq} \in \mathbb{E}$. We need to prove the following for causal consistency to be satisfied:

$$e_q \leq_q e_q^\odot \implies e_p \leq_p e_p^\odot. \tag{3.6}$$

By contradiction, suppose that the following statement is true:

$$e_p^\odot \leq_p e_p \wedge e_q \leq_q e_q^\odot \tag{3.7}$$

Given that $e_p \prec e_q$ and p and q are directly connected processes we can assume that $e_p = \langle send, m \rangle_{pq}$ $e_q = \langle rcvd, m \rangle_{pq}$. Since c_{pq} implements a FIFO Reliable Channel (Specification 2) we know that every message m sent is delivered (FIFORC1) and all messages in a channel maintain FIFO delivery order (FIFORC4).

$$e_p^\odot \leq_p \langle send, m \rangle_{pq} \iff e_q^\odot \leq_q \langle rcvd, m \rangle_{pq} \tag{3.8}$$

If we however infer FIFO delivery order (3.8) on (3.7) we arrive to the following contradiction.

$$e_q^\odot \leq_q \langle rcvd, m \rangle_{pq} \wedge \langle rcvd, m \rangle_{pq} \leq_q e_q^\odot \Rightarrow\!\Leftarrow . \tag{3.9}$$

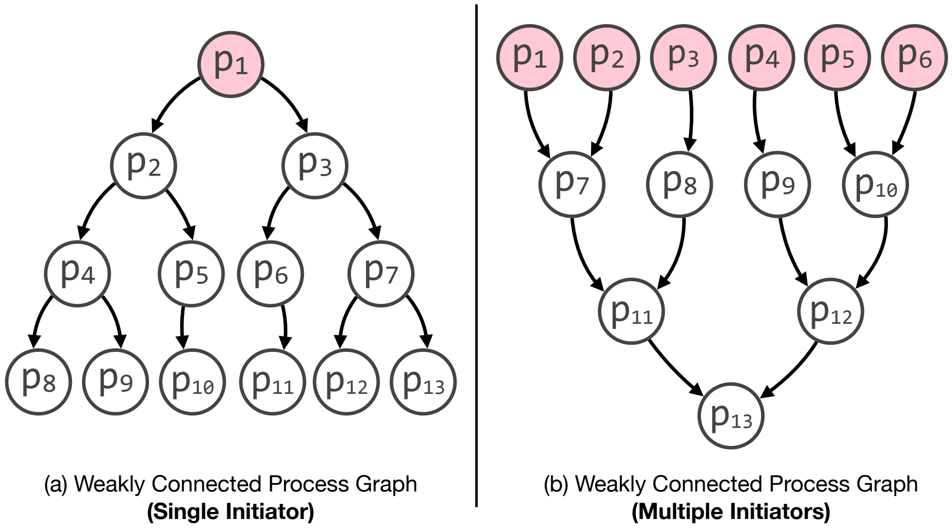


Figure 3.5: Examples of weakly connected graphs with possible protocol initiators.

Finally, by using (3.6) as an induction step we can trivially generalize for any causally related events that can occur along the chain of reachable processes from p to q, $p, q \in \Pi$ where $p \sim q$.

$$\begin{aligned} &(c_{pq} \in \mathbb{E} \wedge e_p \prec e_q) \implies (e_q \leq_q e_q^{\odot} \implies e_p \leq_p e_p^{\odot}) \\ &\iff \ldots \iff \text{(induction)} \\ &(p \sim q \wedge e_p \prec e_q) \implies (e_q \leq_q e_q^{\odot} \implies e_p \leq_p e_p^{\odot} \end{aligned} \tag{3.10}$$

Since we assume a strongly connected graph Equation 3.10 is satisfied for $\forall p, q \in \Pi$, therefore in that case causal consistency can never be violated. Furthermore, in Theorem 3.2.2 we summarize a general observation regarding validity in marker-based snapshotting protocols.

Theorem 3.2.2. Marker-Based Snapshotting: A valid snapshot can be acquired via the use of a marker-forwarding logic in a process graph with FIFO reliable links. According to that logic, a local snapshotting action $e_q^{\odot}$ in a process $q \in \Pi$ is causally related to the snapshotting action $e_p^{\odot}$ of an adjacent process p (i.e., $e_p^{\odot} \prec e_q^{\odot}$) via the the forwarding of a special marker $\odot$ included in that action that separates all events the precede and follow a consistent cut.

$\square$

3.2.4.4 Applicability of Marker-Based Snapshotting

The marker-based protocol of C-L creates a single snapshot and guarantees termination if initiated by a single process in a strongly connected graph. We will now examine alternative protocol initiation strategies that apply to non-strongly connected graphs. As it was shown in the termination analysis in paragraph 3.2.4.2, termination is based on strong connectivity, i.e., $p \sim q$ for any pair of processes $p, q \in \Pi$ (Definition 3.2.5). However, the applicability of the protocol can be generalized further by lifting two of its main assumptions (I) *strongly connected process graphs* and (II) *single initiator*.

(I) Supporting Weakly Connected Graphs

A directed graph is *weakly connected* if by replacing all of its directed edges with undirected ones it produces a strongly connected graph. In principle, any graph that forms a connected component is a weakly connected graph. Consider the process graph of Figure 3.5(a). Since not every process is reachable from any other process (e.g., $p_5 \not\sim p_2$) the marker-based protocol is not guaranteed to terminate if started from any single process. However, in the same process graph $p_1 \sim p, \forall p \in \Pi$ thus, if initiated upon p_1 the C-L protocol will terminate. Lemma 3.2.3 summarizes this observation.

Lemma 3.2.3. Given a weakly connected process graph $G = (\Pi, \mathbb{E})$, a marker-forwarding snapshotting protocol can terminate with a single initiator iff there exists $p \in \Pi$ so that $\forall q \in \Pi$ q is reachable from p (i.e., $p \sim q$).

(II) Supporting Multiple Initiators

We can generalize the applicability of C-L snapshots even further, to any weakly connected process graph for which we can coordinate more than a single initiating process. In the simplest case we can naively initiate the protocol on all processes which is guaranteed to terminate on any graph trivially. However, it is feasible to select a minimal set of initiators $\mathbb{A} \subseteq \Pi$ that can, in combination, reach every other process. This observation follows trivially from Lemma 3.2.3 and generalizes it further since the union of reachable processes via $\mathbb{A}$ yields the remaining processes in the process graph. We summarize this generalization in Definition 3.2.6.

Definition 3.2.6. Given a weakly connected process graph $G = (\Pi, \mathbb{E})$, a marker-forwarding protocol can terminate with any set of initiators $\mathbb{A} \subseteq \Pi$:
$\Pi/\mathbb{A} \subseteq \{q \in \Pi | \exists p(p \in \mathbb{A} \land p \sim q)\}$.

In the example of Figure 3.5(b) consider $\mathbb{A} = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. From Definition 3.2.6 we know that we can reach $\{p_7, p_8, p_9, p_{10}, p_{11}, p_{12}, p_{13}\} = \Pi/\mathbb{A}$. Notice that if we exclude e.g., p_5 from $\mathbb{A}$ then we can only reach $\Pi/\{\mathbb{A}, p_5\}$, however, $\Pi/\mathbb{A} \not\subseteq \Pi/\{\mathbb{A}, p_5\}$. In other words, while the same processes can be reached as

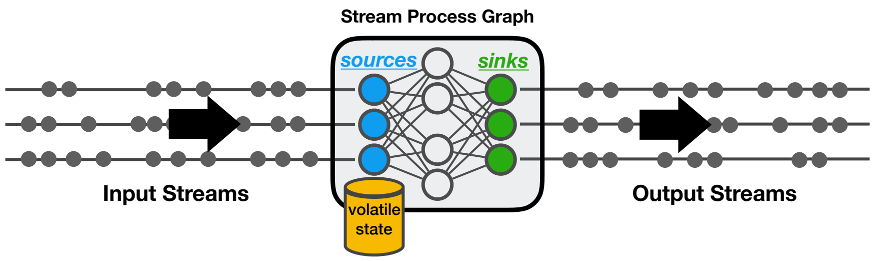


Figure 3.6: A Stream Process Graph.

before without p_5 e.g., $p_6 \sim p_{10}$, no process can reach p_5 itself and thus, p_5 should be part of the initiators in $\mathbb{A}$. In this example, $\mathbb{A}$ is also a minimal set.

3.3 Issues in Reliable Stream Processing

Stateful, stream processing graphs are special-purpose distributed systems that are used for pipelining computational tasks and their interdependencies in continuous, possibly infinite executions. In this section, we present the special properties of stateful stream processing, as well as open problems that can be reduced to the core issue of reliable processing in long-running executions.

3.3.1 Specification of Stateful Stream Processing

First, we make a set of model assumptions for data stream processing which we distill from the common characteristics of known scalable stream processors (e.g., Flink [43] , Apex [39], IBM Streams [56], Storm [5], SEEP [10, 11] and Heron [69]).

3.3.1.1 Stream Process Model

A stream processing system (depicted in Figure 3.6) can be modeled as a special purpose process graph $G = \{\Pi, \mathbb{E}\}$ that, as a single unit, receives ordered sequences of messages as an input, updates its volatile state and generates ordered sequences of output messages. Internally, individual processes that we call "tasks" follow a strict message-driven execution of computational steps, as described already in section 3.2. We further call each step a *stream process action* and denote $\langle \mathrm{proc}, m, M_i \rangle_p$ a process action of a task p that is triggered by an input message $m \in M$ (i.e., $m \neq \{\varnothing\}$) and produces a set of output messages M_i associated with that action.

The internal computation logic within a *stream process action* is encapsulated within a function `process:` $(s_p^i, m_i) \rightarrow (s_p^{i+1}, M_i)$ that typically maps to the invocation of a higher-order function (e.g., map, filter, fold) and its corresponding user-defined function literals (see Flink's model as a reference in chapter 2). More complex functions can be composed on top such as binary stream logic (e.g., join, co-map, co-flatmap) as well as blocking operator logic that is common in stream processing (see stream windows and iterations in Chapters 5 and 6). For example, event-time windows are also invoked via a *stream process action* triggered by watermarks which are input messages, discussed in detail throughout chapter 6. In this chapter, we will consider stream computation at the lowest processing level (i.e., that of a *stream process action* and corresponding events). Finally, we assume that all tasks operate on a fail-stop processing model, and thus, they stop their execution upon any failure and lose their volatile state.

3.3.1.2 Stream Process Graphs

The process graph of a stream processing system, also referred to as "stream process graph" is, in the most basic case, a connected directed acyclic graph[1] which contains two special types of tasks: sources and sinks.

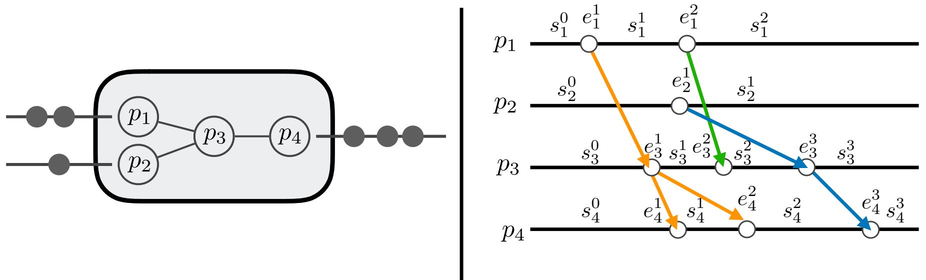
Sources: We term as "sources" a *non-empty* set of tasks that contain no input channels from other tasks and can, in combination, reach all other tasks in the graph. Since sources share the same properties as the initiators in Definition 3.2.6 we will use the same notation $\mathbb{A} \subseteq \Pi$. In practice, sources serve as the main entry point for input streams in the graph and typically bind to message queues that are external to the system. However, it is also possible for sources to generate streams deterministically. In either case, message generation internally occurs within the regular *stream process action* of the sources and follows a strictly deterministic execution. This is achieved either via pulling data from logged input streams (using a partitioned log [51, 70]), a deterministically generated sequence of records (e.g., Fibonacci sequence) or a mixture of logged and deterministically generated input (e.g., records and derived low watermarks).

Sinks: We define another special set $\Omega \subset \Pi$ of tasks that have no output channels to other tasks in G, referred to as "sinks". In practice, sinks serve as an exit point of a stream process graph by pushing sequences of messages outside the system, e.g., to file systems, message queues or DBMSs via external communication mechanisms. Sinks are especially important for committing external side effects (i.e., output message streams) consistently, as it will be shown in chapter 4.

3.3.2 Productions and the Problem of Reliable Processing

An inherent relation between actions on the strictly message-based execution model of stream processing tasks is the one of *productions*. In Figure 3.7(a) we depict a simple process graph consisting of two sources p_1 and p_2 and a single sink p_4. A

[1]section 3.4.4.4 covers cycles as a special case, while chapter 6 elaborates further on semantics and incorporation of arbitrarily nested cyclic computation in stream process graphs.



p_1
p_2
p_3
p_4
s_1^0 e_1^1 s_1^1 e_1^2 s_1^2
s_2^0 e_2^1 s_2^1
s_3^0 e_3^1 s_3^1 e_3^2 s_3^2 e_3^3 s_3^3
s_4^0 e_4^1 s_4^1 e_4^2 s_4^2 e_4^3 s_4^3

(a) A stream process graph (b) An event diagram of a stream computation

Figure 3.7: Example of a stream process graph and a possible execution

corresponding event diagram in Figure 3.7(b) shows a possible execution based on that process graph. Notice that every stream of messages initiated via p_1 and p_2 respectively creates a tree of *stream process actions* often along the whole diameter of the process graph. We call each relation within a tree a "production" and define it further as a partial order relation $\rightsquigarrow$ in a stream execution E in Definition 3.3.1.

Definition 3.3.1. Given an stream process graph execution E and $e, e', e'' \in E$, the *production partial order* relation $\rightsquigarrow$ satisfies the following:

(1) if e and e' correspond to $\langle proc, m_i, M_i \rangle$ and $\langle proc, m_j, M_j \rangle$ respectively where $m_j \in M_i$ then $e \rightsquigarrow e'$.

(2) $(e \rightsquigarrow e') \wedge (e' \rightsquigarrow e'') \implies e \rightsquigarrow e''$ *(Transitivity)*

In Figure 3.7(b) we highlight the transitive closure of each production with a separate color and call each of them, a *production tree*. We can further generalize that each *production tree* is in fact a subset of an execution, related to the transitive closure of an input event (occurring at a source) since no internal events can occur. Furthermore, due to the finite, directed acyclic structure of a stream process graph, productions can only be finite sets, e.g., in the previous example $e_1^1 \rightsquigarrow \{e_3^1, e_4^1, e_4^2\}$ and $e_1^2 \rightsquigarrow \{e_3^2\}$. Event productions imply causality but the inverse is not true. In fact, due to the local event order there is a causal relation between most events that occur from the very beginning of a computation up to its indefinite execution. We can further reduce the causal ordering relation to stream process graphs as defined in Definition 3.3.2 (modification in blue).

Definition 3.3.2. Given an stream process graph execution E and $e, e', e'' \in E$, the *causal partial order* relation $\prec$ satisfies the following:

(1) $e \leq_p e' \implies e \prec e'$.

(2) $e \rightsquigarrow e' \implies e \prec e'$.

(3) $(e \prec e') \wedge (e' \prec e'') \implies e \prec e''$.

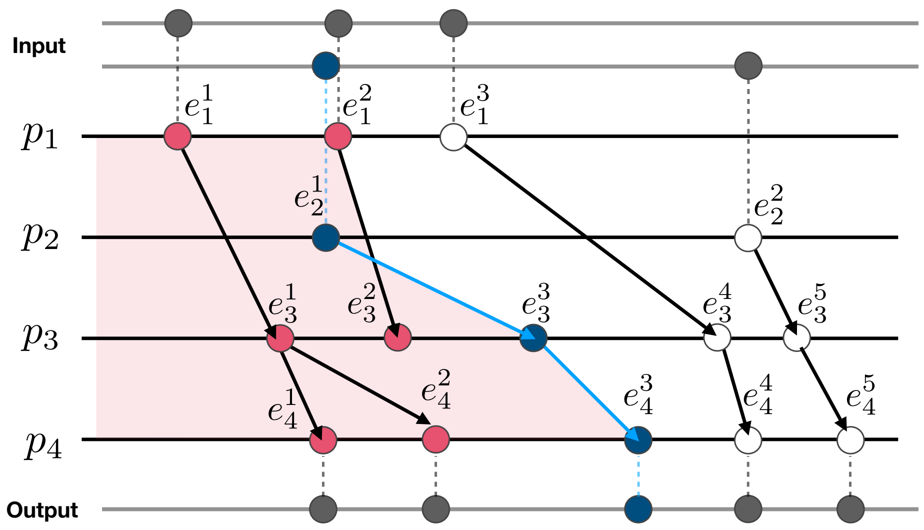


Figure 3.8: A highlight of all events that produce (blue) and cause (red) e_4^3.

To visualize the two relations, consider a continuation of the previous execution of Figure 3.7(b) including mappings of source and sink events to input and output streams as depicted in Figure 3.8. A selected event e_4^3 is part of the production tree of e_2^1 since $e_2^1 \rightsquigarrow \{e_3^3, e_3^4\}$ and causally succeeds most events of the distributed execution up to this point (as shown in red in the figure). It is noticeable that throughout an execution every individual event contributes to side-affects both to the stream output with its productions but also to all causally succeeding events via internal state transitions.

Problem Intuition: Given the unbounded nature of stream processing executions and the large number of tasks allocated in data centers, we expected failures to occur especially often during the lifetime of an application. If we ought to strongly rely on such an execution (e.g., processing medical records or financial transactions) and the correctness of its corresponding results we have to ensure that the system hides inconsistencies from the outside observer. This means the observed system execution (inferred by looking at each individual state transition) should hide events that correspond to incomplete or abnormal state that might have occurred at its tasks (e.g., incomplete productions, undetected event patterns etc.). Before providing a clear definition of the problem of reliable stream processing we will first retrospect over the state of the art as well as the basic modern needs for reliable processing.

3.3.3 Reliable Stream Processing: Past and Current Considerations

While in our design we have adopted a fail-stop failure model, there have been several approaches in that past that assume a fail-recovery model, where tasks can restart independently from failures and employ different mechanisms [63, 62, 9, 11, 10] to amend their execution to reproduce computations that were possibly lost. We briefly discuss two such related approaches and their considerations.

Task-Level Rollback Recovery: One approach to provide fault tolerance as well as aid reconfiguration is to allow processes to checkpoint independently their channel and internal states to stable storage during arbitrary points in their execution and recover from there independently. However, this creates several non-trivial complications. Consider in the previous example of Figure 3.8 a failure of task p_3 right after executing event e_3^3.

Assuming a fail-recovery model, p_3 could restart from a previously captured state, e.g., s_3^1 and it would have to ensure that 1) pending messages are re-executed strictly in the same order (i.e., processing the message sent from p_2 after the two messages sent by p_1) and 2) consumer tasks do not execute duplicate computation. In our example, task p_4 would have to avoid processing duplicate messages since that would result in an abnormal execution that does not reflect the case of no failures occurring (e.g., if e_4^3 has already already occurred). Several existing approaches consider extensive input logging and complex reconciliation protocols with reachable processes [11, 71, 62] in the graph that is often infeasible with one-directional communication. Furthermore, given the fine-grained nature of task-level rollbacks, from the user- or consuming service-level there is no clear distinction on which parts of the global computation have been committed, thus, no clear guarantees can be made trivially on the state of the application and globally committed output.

Transactional Productions and Idempotency: Another related approach to the problem is to commit each single production as a transaction using an external fault-tolerant storage system, before delivering it to a consumer service or user. Google's Millwheel stream processing system [9] is using BigTable[72] for that purpose which provides support for distributed transactions, committing a production [2] of an input record carrying a specific key with the same transaction identifier (for idempotency). This works trivially in purely data-parallel computations (per-key) where task-level causal order can be possibly violated and all that is needed is to guarantee that each operation per key has succeeded. However, since the more general stream processing model presented here addresses the complete state of each task (across all keys) this approach cannot be generally applicable. Furthermore, the same approach suffers from non-trivial optimisations (e.g., deduplication, batching

[2]Millwheel calls this concept a strong production and differentiates it from weak productions that related to task-level rollback recovery and avoid pre-commits for naturally idempotent productions.

transactions) and also falls short of offering broader application-level guarantees if anything more than committed aggregates per-key are needed.

3.3.4 Related Open Problems

Evidently, there is a need for an application-level consistency model for reliable data stream processing that goes beyond the scope of task or procuction-level fault tolerance. We have identified a set of emerging system needs that relate indirectly to the same problem but cannot be collectively covered or composed by prior approaches.

Support for Reconfiguration: The stream process graph G as well as user-defined logic (e.g., function literals) that is being executed within the tasks is not necessarily static. Given that a stream system does not typically run for a few minutes or hours but is meant to do continuous processing it is often required to apply system-level changes. For example, there is commonly a need to increase or decrease the number of task instances that execute a logical operation in parallel, also known as scale-out or scale-in operation in a cloud computing infrastructure. This problem also demands flexibility on how state is partitioned and re-allocated to different numbers of tasks. Changes in task logic (the *process* function) is also a common case of reconfiguration, associated with "bug fixing" or issuing upgrades on the application logic ran by specific tasks. The main challenge of any type of reconfiguration, is to apply all changes in a reliable and transparent way, similarly to fault recovery.

Execution Migration and Provenance: Stream processing executions might be long-running, however, this is not always the case with the underlying infrastructure. Given the vast amount of diverse IaaS platforms for both on-premise and cloud deployments there is an increasing need for flexibility on *where* an execution is physically occurring. For a reliable stream processing system that means that there is an inherent need to first identify, capture and version the global state of a stream execution e.g., E^{v1} in order migrate to E^{v2} where $E^{v1} \subseteq E^{v2}$. To enable this, all production trees would need to maintain causal dependencies from E^{v1} to E^{v2}.

State Access Isolation: Following the recent trend of main-memory databases, many latency-critical applications today rely on fast read access. The state of of individual processors in stream processing graphs poses an attractive alternative to main-memory databases [73], since among other things, it always reflects an up-to-date summary of long sequences of input records. Given that this state unlike DBMSs is not transitionally committed it is restricted, by definition, to dirty reads. That yields a need for a form of version control for task states in stream processing, separating state that is potentially safe to read from volatile state that can be invalidated. That can further allow the possibility of access isolation mechanisms that can guarantee causal consistency to external queries.

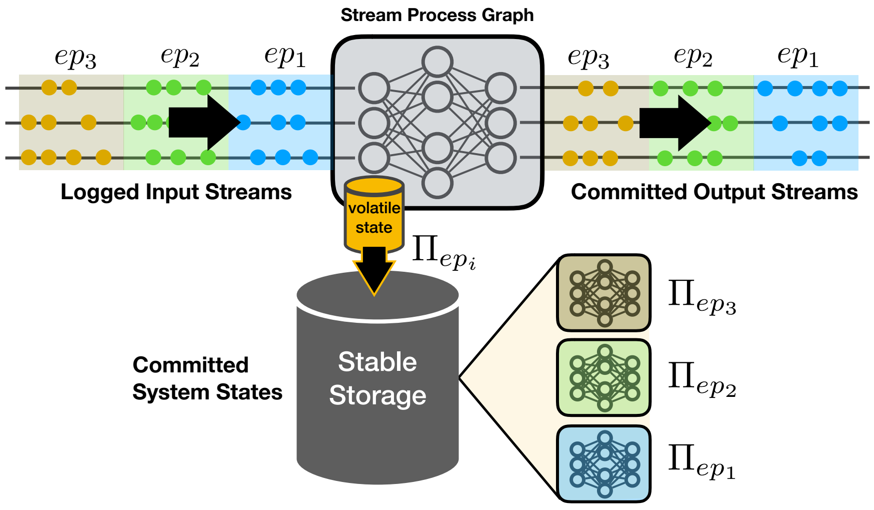


Figure 3.9: Overview of Epoch-Based Stream Processing.

3.4 Epoch-Based Reliable Stream Processing

The concept of epoch-based stream processing offers a uniform solution to reliable streaming and fulfills all relevant state management needs discussed in subsection 3.3.4. In this section, we introduce the concept and describe several design approaches to materialize it, while aiming to satisfy principal design properties (uncoordinated and blocking-free execution, transparency and compositionality).

3.4.1 Concept Overview

In Figure 3.9 we visualize the essential idea of epoch-based processing at a conceptual level. We assume, as before, a stream process graph G with a fail-stop model and a deterministic event sequence at its sources (e.g., logged streams). The main goal is to expose a continuous execution that satisfies reliable processing, i.e., maintaining all event productions and respecting causality. However, instead of reasoning about individual events, we distinguish discrete stages, or "epochs" that represent finite intermediate subsets of a continuous execution E as such $\mathsf{E}^{ep_1} \subset \mathsf{E}^{ep_2} \subset \mathsf{E}^{ep_3} \ldots \subset \mathsf{E}$.

Conceptually, each epoch represents a part of the computation that is atomically processed (either completes or restarts). This allows us to lift all concerns regarding reliability from the level of individual records or productions to the coarse-grain level of epochs. Once an epoch has been committed all the states of the process

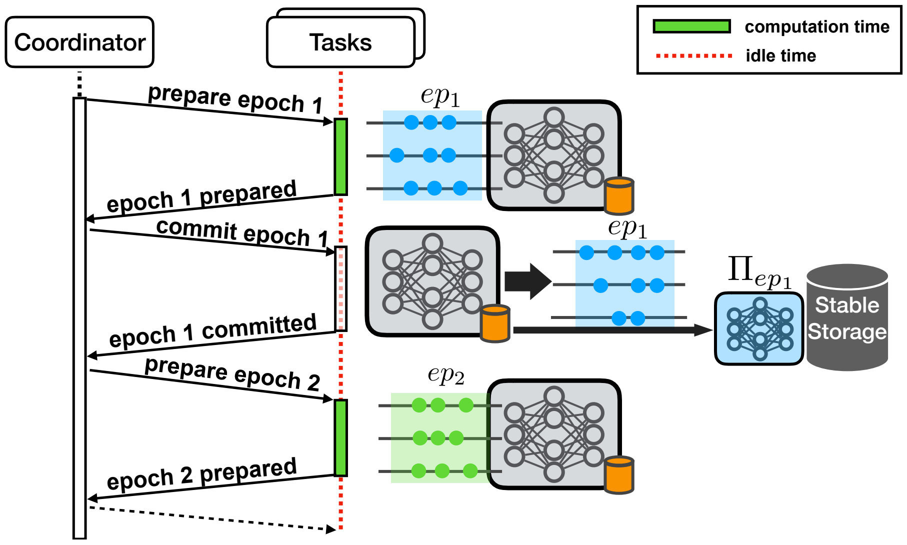


Figure 3.10: Example of Synchronous Epoch Commit.

graph in addition to external output streams reflect the full state of an execution up to that point. In case of failure during the execution of an epoch ep_n we can simply rollback the deterministic input and global state of the process graph to a previously completed epoch (e.g., ep_{n-1}). A core requirement for employing an epoch-based execution is to be able to capture all individual task states of the system upon the completion of an epoch on stable storage. It is, in essence, the configuration of the system at the exact point that the input of epoch ep_i (and nothing more) has been fully processed. Thus, a snapshot S_{ep_i} of that configuration would consists only of local task states $S_{ep_i} = \{\Pi_{ep_i}\}$. It is important to note here that there are many possible executions up to an epoch completion, e.g, $E_{ep_i} \neq E'_{ep_i}$, since no total processing order is guaranteed in a distributed stream execution. However, a committed epoch includes the result of a single possible execution up to an epoch. This notion relates to the informal use of *exactly-once* [9, 29] end-to-end processing semantics which, in this case, it specifically refers to "exactly one" epoch commit.

3.4.2 Synchronous Epoch Commit

In the simplest case, epoch-based stream processing can be achieved via staging the underlying execution. Staging is typically implemented using an atomic commit protocol between a coordinator process and the tasks of the graph.

In Figure 3.10 we show how a synchronous (two-phase) atomic commit protocol can be used to stage epochs in a stream processing execution. The first phase ensures that all computation has been completed and the second phase completes once all side effects have been persisted to stable storage (i.e., task states and stream output). While this approach works in practice, there is one major downside, it enforces a blocking coordination protocol. The core of the problem of staging is that most tasks will have to remain idle 1) until every other task has finished computing and 2) while every other task commits its state and side effects to stable storage. The latter can be potentially avoided with asynchronous copying methods, however, if epochs are frequent the communication overhead of the protocol itself can often overtake the actual computation time. On the other hand, if epochs are infrequent, tasks are less idle but the end-to-end latency between the ingestion of input streams and the time results are committed can be too high and thus, too late for many applications (e.g., critical event monitoring).

3.4.2.1 The Case of Micro-batching

Stream micro-batching [12] is a mechanism that emulates stream processing capabilities on batch processing systems which employ short-lived task execution (i.e., Spark [27], MapReduce [8]). In essence, micro-batching is equivalent to the synchronous epoch commit protocol. The main idea is to have a coordinating process (i.e., the driver in Spark) that periodically divides a stream into batches and then schedules a job (process graph of short-lived tasks) per batch. In the case of micro-batching, a new set of tasks will be scheduled to overtake the computation on each epoch. In case of a failure, a batch is re-executed by the same or a new set of tasks (parallel recovery[12]). This grants the flexibility to re-allocate and reconfigure the computation per batch, however, it also introduces additional scheduling overhead on top of the synchronous epoch commit costs mentioned before. Furthermore, the original version of discretized streams enforced the use of time-discretized batches in its programming model, making the whole architecture non-transparent to the user-facing programming model. This problem can be attributed to an initial design choice in the context of batch execution rather than being an inherent property of epoch-based stream processing, as newer versions of Spark Streaming offer a more fluid stateful programming model (e.g., Structured Streaming [74]).

3.4.3 Asynchronous Epoch Commit

We seek for non-blocking mechanism for committing epochs in a stream computation, one that does not halt the regular execution of the stream processing graph and allows tasks to progress asynchronously across epochs without the need to remain idle for blocking synchronization. A possible solution to satisfying this fundamental requirement lies at the use of consistent snapshots, regulated to respect epochs in order to bypass the need for staging the overall execution.

3.4.3.1 Epoch Events and Cuts

First, we will examine the notion and implications of epochs in a continuous execution. An epoch marks a logical time for a distributed stream computation known at all sources of a stream process graph. We will denote as ep_i the epoch of time $i \in \mathbb{Z}$ and further assume that sources are notified about the completion of an epoch, e.g., ep_n through special *epoch events* : $\langle ep_n \rangle$.

Epoch events can be either issued by an external coordinator process to all source tasks, or instructed by a user or simply inferred through special punctuation events pulled from incoming data streams. For generality, we will make no special assumptions at this point on how epoch events are triggered, though, we will assume that they are eventually triggered at all sources $\mathbb{A} \subseteq \Pi$ of a stream process graph and are monotonic (given in Definition 3.4.1).

Definition 3.4.1. Epoch Event Monotonicity: Given a source task $p \in \mathbb{A}$ and two local epoch events in E_p: $e_p^k = \langle ep_i \rangle$ and $e_p^l = \langle ep_j \rangle$ then $i < j$ iff $k < l$.

The notion of epoch events can help us reason about a complete preceding computation in a stream process graph. In Definition 3.4.2, we introduce "Epoch Cuts", a stricter form of a consistent cuts expressing not only a causally consistent, but also a *complete* execution of a stream process graph in respect to an epoch.

Definition 3.4.2. Epoch Cut: An Epoch Cut $\mathcal{C}_{ep_i}$ satisfies the following invariants in stream process graphs (acyclic) for $e, e' \in E$ and $p \in \mathbb{A}$:

(1) $(\langle ep_i \rangle \prec e) \implies (e \notin \mathcal{C}_{ep_i})$

(2) $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$

(3) $((e \in \mathcal{C}_{ep_i}) \wedge (e \rightsquigarrow e')) \implies (e' \in \mathcal{C}_{ep_i})$

(4) $((e \in \mathcal{C}_{ep_i}) \wedge (e' \prec e)) \implies (e' \in \mathcal{C}_{ep_i})$ (consistent cut)

We will be using invariant (4) in order to describe *Causal Consistency* and (1-3) as the necessary conditions for *Epoch Completeness* in a cut.

Example: Consider the execution illustrated in Figure 3.11 which is based on the process graph of Figure 3.1 and includes epoch events for ep_n. The events marked as green denote everything that locally precedes epoch events, including their productions, while all succeeding production trees are marked as red. Based on Definition 3.4.2, the epoch cut $\mathcal{C}_{ep_n}$ should contain exactly all green events including the epoch events. Both C_1 and C_2 satisfy *Causal Consistency*, yet, only C_2 satisfies *Epoch Completeness* and therefore, $C_2 = \mathcal{C}_{ep_n}$. In the case of C_1 we identify two invariant violations for Epoch Cuts. First, events $e_1^4, e_2^3 \in C_1$ while $\langle ep_n \rangle \prec_{p1} e_1^4$ and $\langle ep_n \rangle \prec_{p2} e_2^3$ which violates invariant (1) of Definition 3.4.2. Second, $e_4^3 \notin C_1$, however, according to Definition 3.4.2(3) it should be part of the epoch cut since $(e_1^2 \in C_1) \wedge (e_1^2 \rightsquigarrow e_4^3)$. Therefore, $C_1 \neq \mathcal{C}_{ep_n}$.

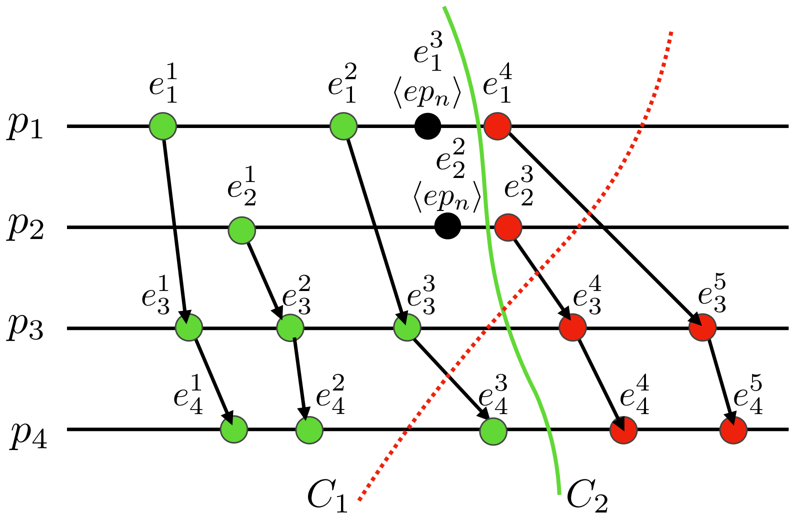


Figure 3.11: An example of two consistent cuts where $C_2 = \mathcal{C}_{ep_n}$.

3.4.3.2 Feasibility of Epoch Cuts

Evidently, epoch cuts are a quite strict form of consistent cuts since they impose constraints on the possible executions where they can be feasible. In Figure 3.12 we show another possible execution based on the process graph of Figure 3.1 as before. The problem in this execution is that no consistent cut can satisfy all invariants of Definition 3.4.2. For example, by invariant (1) and (4) it should be true that $e_3^4 \in \mathcal{C}_{ep_n}$ since $e_1^2 \prec_{p1} \langle ep_n \rangle$ and $e_1^2 \rightsquigarrow e_3^4$. However, this contradicts with $e_3^4 \notin \mathcal{C}_{ep_n}$ imposed by invariant (2) since $\langle ep_n \rangle \prec e_3^4$. (i.e., $\langle ep_n \rangle \prec e_2^3 \prec e_3^3 \prec e_3^4$).

Intuitively, in order to make any epoch cut feasible, an execution E should always respect event-ordering imposed by epochs. Thus, no event that is not part of an epoch should precede an event of that epoch. We call this execution property "Epoch Feasibility" and summarize it in Definition 3.4.3.

Definition 3.4.3. Epoch Feasibility: An execution E of a stream process graph satisfies Epoch Feasibility iff $\forall e, e' \in E : ((e \in \mathcal{C}_{ep_n}) \land (e' \notin \mathcal{C}_{ep_n})) \implies e' \nprec e$.

3.4.4 Snapshots for Asynchronous Epoch Commit

If we can assure the feasibility of epoch cuts throughout a long-running continuous execution then it is possible to prepare and commit epochs asynchronously, as depicted in Figure 3.13. This can be enabled through the use of snapshots, though, we need a protocol that implements not just any consistent cut but epoch cuts.

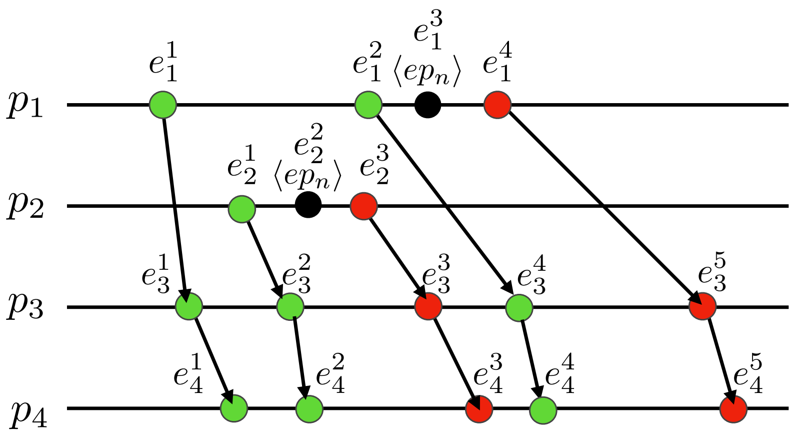


Figure 3.12: An example of an execution where $\mathcal{C}_{ep_n}$ is infeasible.

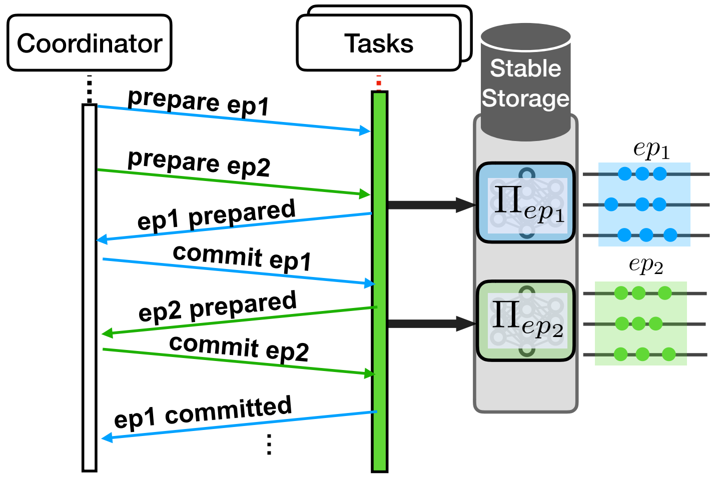


Figure 3.13: Asynchronously coordinated epochs with no idle times.

Specification 4 describes the expected behavior of "Epoch Snapshotting". Epoch snapshots are consistent snapshots that capture an *epoch-complete* configuration $S_{\mathcal{C}_{ep_n}}$ in respect to an epoch cut $\mathcal{C}_{ep_n}$. The specification essentially describes a recurring consistent snapshot that terminates per epoch in a continuous execution. When it comes to safety properties an epoch snapshot should satisfy *Validity* (identical to the consistent snapshot specification) as well as *Epoch Completeness*.

With epoch completeness we refer to the notion of capturing the (local) execution of an epoch ep_n in a process p, also denoted as $E_p^{ep_n}$. Given that every full epoch-complete snapshot implements an epoch cut, all message productions are included in the snapshot and therefore, no messages in transit are considered in a captured configuration. Mind that in this specification snapshots are not single-shot but recurring and should satisfy *Termination* if no failures occur up to the complete execution of an epoch.

Specification 4: Epoch Snapshotting ($\mathtt{esnap}$)

Event Interface:

Indication: $\langle \mathrm{record} | p, n, s_p^i \rangle$

Properties:

ESNAP1: *Termination*: $\Lambda = \Pi$ in $E^{ep_n} \implies \langle \mathrm{record} | p, n, \_ \rangle \in E_p, \forall p \in \Pi$

ESNAP2: *Validity*: Configuration $S_{\mathcal{C}_{ep_n}} = \{\Pi_{ep_n}\}$ is valid

ESNAP3: *Epoch-Completeness*: Configuration $S_{\mathcal{C}_{ep_n}}$ is epoch-complete

Snapshotting Approach: The in-flight marker-based snapshotting mechanism by Chandy-Lamport that we analyzed previously in subsection 3.2.3 demonstrates how to capture a consistent snapshot concurrently to the execution without the need for blocking coordination. However, in the context of epoch-based processing it is not capable of offering epoch-cuts as is. In the rest of this section, we present step-by-step an alternative marker-forwarding protocol that satisfies validity (Theorem 3.2.2) similarly to Chandy Lamport while also guaranteeing epoch completeness. For better understanding, we break down our approach into three parts: 1) Snapshot Initiation, 2) Epoch Alignment and 3) Cyclic State.

3.4.4.1 I) Snapshot Initiation

The Termination property of $\mathtt{esnap}$ hints that the protocol has to eventually yield a snapshot per epoch. Therefore, we need a mechanism that initiates a snapshot per epoch and eventually terminates for every task in Π. In Theorem 3.2.6 we generalized the applicability of marker-forwarding snapshotting protocols to any weakly connected graph. More concretely, we have proven that the marker-based protocol terminates if initiated on a set of processes $\mathbb{A}$ that can in combination reach the full graph. In the context of stream process graphs, the set of source tasks satisfies this property. That means that an instance of the marker-based protocol would eventually reach all processes if executed on a set of initiator processes A.

Algorithm 5: Epoch-Based Snapshots (Sources)

```
Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{I}_p, \mathbb{O}_p$)
Algorithm:
1: $\mathbb{O}_p \leftarrow$ configured_channels;
2: $s_p \leftarrow \varnothing$;                                   ▷ volatile local state
3: /* Source Task Logic                                           */
4: Upon $\langle rcvd, m \rangle$
5:     $(s_p) \leftarrow process(s_p, m, \mathbb{O}_p)$;
6: Upon $\langle ep|n \rangle$
7:     $esnap \rightarrow \langle record|self, n, s_p \rangle$;
8:     foreach $out \in \mathbb{O}_p$ do
9:         $out \rightarrow \langle send, \odot_n \rangle$;
```

Given that every source task eventually becomes aware of an epoch change at the moment it processes an epoch event (e.g., $\langle ep_n \rangle$ for epoch n), it is natural to also initiate the protocol at that very instant. In algorithm 5 we summarize the epoch-based snapshot initiation logic for the source tasks. Mind that despite the fact that source tasks always follow a strictly deterministic order of events, they do have internal state. The internal state of the sources typically encapsulates the exact point in their deterministic execution. For example, if a source task reads messages from a logged message queue, its state is the read offset or pointer in that queue. Furthermore, we attach the epoch number to markers, in order to make the current epoch known to all receiving tasks and later simplify the collection of states based on their respective epoch that they were captured. Overall, given that source tasks have no input channels, the logic here is identical to that of the C-L algorithm.

3.4.4.2 II) Epoch Alignment

As we have concluded in Theorem 3.2.2, the marker-based "snapshot-and-forward" logic can guarantee validity. It remains to examine the necessary modifications needed to satisfy epoch completeness. In Figure 3.14 we depict a possible cut consistent and associated (valid) snapshot based on the previous example of an execution where epoch cuts are infeasible. Mind that message m is captured as part of the snapshot. Instead, an epoch-complete cut should incorporate all remaining productions $e_1^2 \rightsquigarrow \{e_3^4, e_4^4\}$. We can do so via a form of prioritization that we call "epoch alignment".

The epoch alignment mechanism allows tasks to complete all computation associated with an epoch before proceeding further in an orthogonal manner to

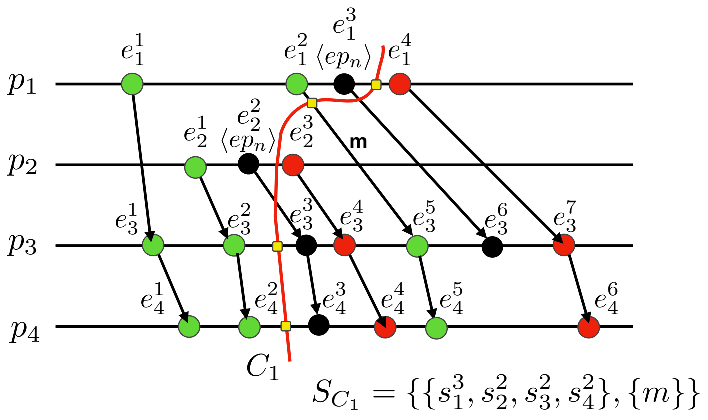


Figure 3.14: A consistent but not epoch-complete snapshot using C-L.

Algorithm 6: Epoch-Based Snapshots (Regular Tasks)

```
Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($\mathbb{I}_p, \mathbb{O}_p$)
Algorithm:
1: ($\mathbb{I}_p, \mathbb{O}_p$) ← configured_channels;
2: Enabled ← $\mathbb{I}_p$ ;
3: $s_p$ ← ∅;                                              ▷ volatile local state
4: /* Common Task Logic                                                       */
5: Upon ⟨rcvd, m⟩ on c ∈ Enabled
6:     $s_p$ ← process($s_p$, m, $\mathbb{O}_p$);
7: Upon ⟨rcvd, $\odot_n$⟩ on c ∈ Enabled
8:     esnap → ⟨record|self, n, $s_p$⟩;
9:     Enabled ← Enabled/{c};
10:    if Enabled = ∅ then
11:        foreach out ∈ $\mathbb{O}_p$ do
12:            out → ⟨send, $\odot_n$⟩;
13:        Enabled ← $\mathbb{I}_p$ ;
```

the marker-forwarding logic that guarantees validity. The goal is to turn any execution into a feasible one for epoch-cuts and therefore, obtain a snapshot that is both causally consistent (valid) and epoch-complete. In algorithm 6 we describe the complete logic of epoch alignment, as a minor modification of the core C-L protocol. Epoch markers are again disseminated throughout the graph, though, with alignment we first make sure that all markers have been received along the inputs before capturing the task state and disseminating further.

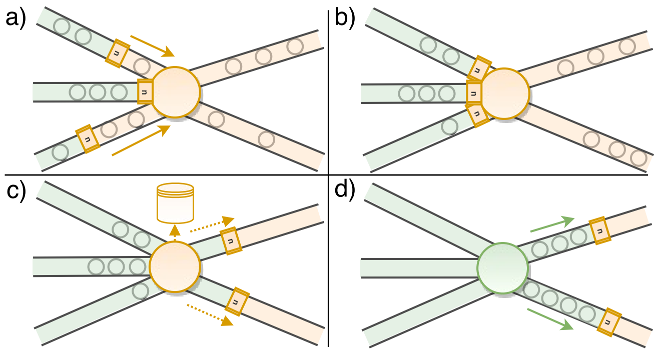


Figure 3.15: Alignment and Snapshotting Highlights.

Figure 3.15 depicts the steps prior to and during snapshotting in more detail. When a task receives a marker on one of its channels, it removes that channel from the "enabled" ones since all computation associated with the current epoch has to complete before continuing further (Figure 3.15(a)). Once markers have been received in all inputs (Figure 3.15(b)) the task can further capture its full state and notify downstream tasks based on the marker-forwarding logic that maintains causal consistency along the recording process of the global snapshot in the graph.

Example: In Figure 3.16 we visualize how epoch alignment can make epoch cuts feasible in the context of the execution visited previously. Alignment takes place in task p_3 once it processes the epoch marker in event e_3^3. At that point e_{23} is removed from the pending channels of p_3, prioritizing messages through e_{13}. Eventually, the last epoch marker is being processed in e_3^5 resulting into p_3 dropping alignment, snapshotting its state and propagating the epoch marker further to p_4. The resulting cut C_2 satisfies all properties of an epoch cut and thus $C_2 = \mathcal{C}_{ep_n}$.

3.4.4.3 Alignment vs Synchronous Epoch Commit

Epoch alignment effectively results into equivalent executions as the synchronous epoch-based approach (commit per epoch). However, in contrast to the synchronous approach, alignment does not hinder blocking synchronization since all tasks continue their regular operation, while prioritizing pending work to complete an epoch in a coordination-free fashion. The sole cost of alignment is limited to extra in-transit latencies for messages within non-enabled channels (measured further in chapter 4 section 4.4). In a typical push-pull messaging model that is employed in most stream processing systems, channel omission can lead, in the worst case, into disk spilling when allocated memory for network buffers reaches its limit. However,

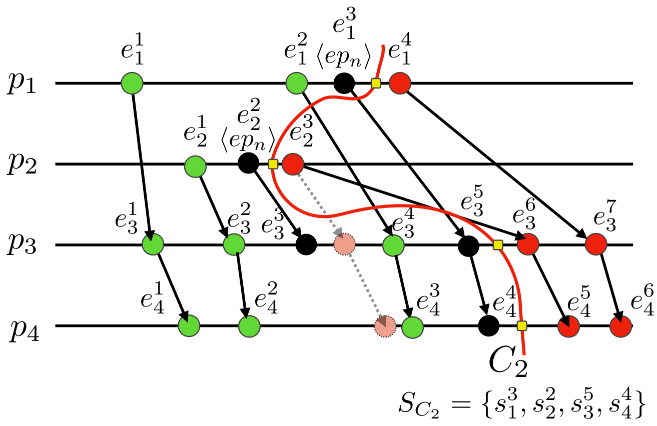


Figure 3.16: An epoch-complete snapshot using epoch alignment.

credit-based flow control can also be issued in order to avoid pushing messages on channels that are non-pending from receiving tasks.

3.4.4.4 III) Cyclic State

So far we have only considered the case where the stream process graph is a directed acyclic graph. However, it is often required to incorporate closed loops of tasks in stream processing graphs (e.g., in machine learning and graph processing applications). We identify two main challenges related to cyclic graphs : 1) A production tree within a cycle is not guaranteed to be bounded and thus, it can potentially progress infinitely. That would in turn mean that an epoch is not guaranteed to complete. 2) Epoch alignment deadlock as-is if there existed a loop in the graph. That is due to the fact that the protocol progresses only if all preceding computation on an epoch has completed. Hence, a circular dependency would make such a condition unsatisfiable.

Given that explicit loops disallow epoch-based execution we can only deal with such issues indirectly. Consider a process graph with cycles, such as the one depicted in Figure 3.17. Loops are formed by strongly connected components in a stream process graph. Conceptually, if we collapse each strongly connected component into a single task (e.g. loops A and B in Figure 3.17) we get a directed acyclic graph. Assuming that we are able to employ any epoch-based execution in that conceptual level, the complete state of Loop A and B at the pass of an epoch would in fact be their in-progress computation, a combination of internal states and in-transit messages. This concept encapsulates our approach to dealing with loops which we describe in a methodology comprising of two steps: 1) Back-Edge Identification and 2) Loop Expansion, as described below. Our methodology is general and it can be used to break loops in any strongly connected component.

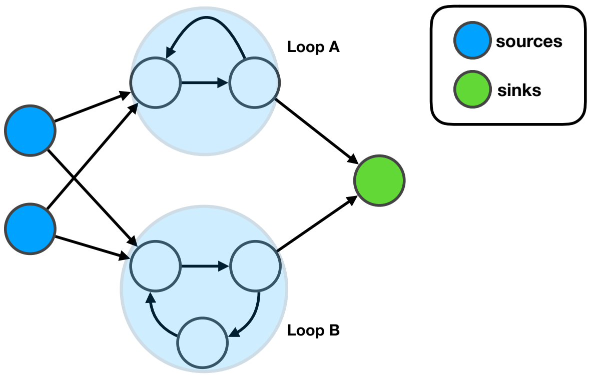


Figure 3.17: An example of a process graph with two loops.

Back-edge Identification: In the best case, loops can be explicitly defined at a higher level programming model (see chapter 6). However, there also exist compositional dataflow programming systems that allow loops to be composed arbitrarily (e.g., Apache Storm [38]), similarly to go-to statements in imperative programming. In the most general case each loop can be inferred using a typical strongly connected component detection algorithm [75] in a depth-first traversal of the process graph at compilation time. Given a set of tasks per loop it is then possible to define an entry task based on the highest dominance value [76]. Typically, in stream process graphs the entry task corresponds to the one that maintains the most input edges external to the loop (since all stream flow passes through that node). An edge (channel) within the cycle that points to the entry task is also known as a back-edge. In Figure 3.18(a) we highlight in red each selected back-edge per loop.

Loop Expansion: Given that we all back-edges are identified, the next step is to eliminate cycles via a process we call "Loop Expansion". We replace each back-edge with two tasks, a source task called "Loop Head" and a sink task called "Loop Tail". Each pair of Head and Tail is interconnected with a hidden channel that we call phantom channel. All messages that traverse the phantom channel are processed by the Head as regular input records. This way our graph is a directed acyclic graph and can respect epochs since each Loop Head is able, as every other regular source to process epoch events and therefore determine which part of the in-transit (cyclic) computation belongs to an epoch. This approach works seamlessly for nested loops as well since every cycle will be transformed into an acyclic graph with a source and a sink, eliminating any cyclic dependencies and allowing epoch-based progress.

The Loop Head tasks incorporate our final variant of the epoch-based snapshotting protocol, summarized in algorithm 7 which guarantees to capture the current state of each loop (messages in transit) at the moment an epoch changes. The overall logic is identical to Chandy-Lamport snapshots since all in-transit messages are

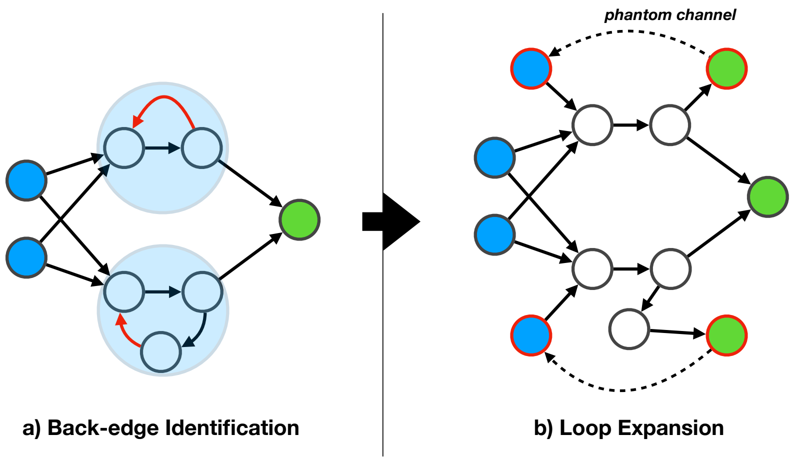


Figure 3.18: Process Graph Transformation Steps for Loops

recorded up to the completion of an epoch. However, we only apply this logic at Loop Head nodes, where that functionality is necessary in the context of a strongly connected component.

We visualize each step of this algorithm variant in Figure 3.19, starting at the instant that a loop head task receives an epoch change event. As depicted in Figure 3.19(a)), markers are first inserted within the respected cycle through the head task. From that point until a marker is processed back at the head (after traversing the whole cycle) all messages forwarded at the head through the phantom channel are being also logged into the head's snapshotted state since they precede the marker and therefore belong to preceding epochs (similar to the C-L algorithm). Once the marker is received back at the head (Figure 3.19(c)) its message log is committed as part of its internal state and the protocol terminates (Figure 3.19(d)).

3.4.5 Analysis of Asynchronous Epoch Snapshots

The epoch-based snapshotting protocol presented inherits the core invariants of the C-L protocol when it comes to termination and validity, while also satisfying epoch completeness. In this section, we will examine these properties and prove them in the context of epoch-based stream processing.

3.4.5.1 Termination

In section 3.2.4.2 we have proven the termination property of the marker-forwarding logic in the C-L protocol which we further generalized to weakly connected graphs with multiple initiators in Definition 3.2.6. According to these observations, any

Algorithm 7: Epoch-Based Snapshots (Loop Heads)

```
Implements: Epoch-Based Snapshotting (esnap)
Requires: FIFO Reliable Channel ($(\mathbb{O}_p)$)
Algorithm:
1:  $(\mathbb{O}_p) \leftarrow$ configured_channels;
2:  recording $\leftarrow$ false;
3:  $s_p \leftarrow \emptyset$;                                  ▷ logged in-transit state
-----------------------------------------------------------------
4:  /* Loop Head Logic                                          */
5:  Upon $\langle ep|n \rangle$
6:      recording $\leftarrow$ true ;
7:      foreach out $\in \mathbb{O}_p$ do
8:          out $\longrightarrow \langle send, \odot_n \rangle$;
9:  Upon $\langle rcvd, \odot_n \rangle$ on phantom_channel
10:     esnap $\longrightarrow \langle record|self, n, s_p \rangle$;
11:     $s_p \leftarrow \emptyset$;
12: Upon $\langle rcvd, m \rangle$ on phantom_channel
        if recording then
            $s_p \leftarrow s_p \cup m$;
        foreach out $\in \mathbb{O}_p$ do
13:         out $\longrightarrow \langle send, m \rangle$;
```

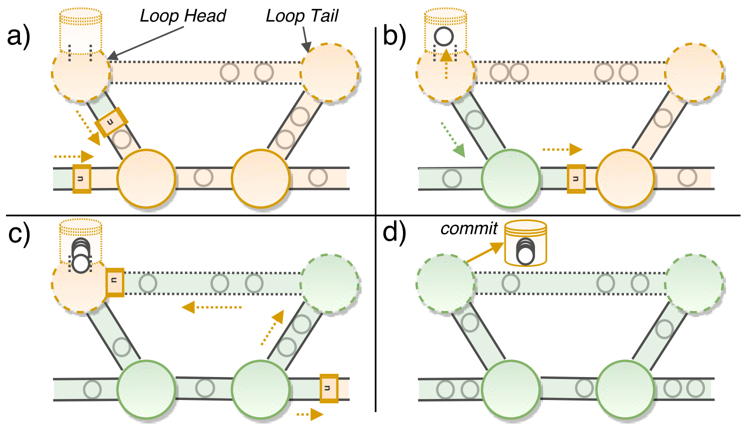


Figure 3.19: Cycle Snapshotting Highlights.

marker-forwarding protocol for snapshotting can terminate if a marker reaches all channels in a process graph. The alignment logic that we have added for epoch completeness affects only prioritization across channels and does not alter the fact that a marker will be in fact delivered across a channel if it has been previously sent (including the phantom channel we have introduced in loops).

Proof. It suffices to show that the following two conditions are satisfied:

1 There exists a set of initiating processes $\mathbb{A} \subseteq \Pi$:
$\Pi/\mathbb{A} \subseteq \{q \in \Pi | \exists p(p \in \mathbb{A} \wedge p \sim q)\}$

2 The protocol is instantiated on every initiating process per epoch.

Condition 1. is satisfied by the properties of a stream process graph itself, including the transformations we introduced for strongly connected components. Effectively, the set of regular sources and the loop heads satisfy the necessary reachability condition.

Condition 2. is satisfied by Algorithms 5 and 7 that describe the initiating logic in both regular sources and loop heads. Given that an epoch event is guaranteed to be received during an epoch change $\langle ep|i\rangle \in E_p, \forall p \in \mathbb{A}$, the algorithm proceeds with the regular marker-forwarding logic within the same action. Furthermore, based that monotonicity property of epoch events given in Definition3.4.1 it is guaranteed that the protocol will be initiated and terminated in strict epoch order.

Finally, we should add that similar to the C-L algorithm, the termination of an instance of the protocol can only be guaranteed if all of the participating processes are correct during the system execution, up to its completion. □

3.4.5.2 Validity

Despite the addition of the alignment mechanism, epoch-based snapshots preserve the same marker-forwarding logic of C-L according to Theorem 3.2.2 and also rely on FIFO reliable channels to guarantee validity.

Proof. Each local snapshotting action $e_q^{\odot}$ in a process $q \in \Pi$ is causally related to the snapshotting action $e_p^{\odot}$ of an adjacent process p in a stream process graph (i.e., $e_p^{\odot} \prec e_q^{\odot}$) via the the forwarding of a special marker $\odot$ included in that action that separates all events the precede and follow a specific epoch cut. In Section 3.2.4.3, we have proven that this principle is equivalent to maintaining causal consistency during snapshot acquisition (summarized in Theorem 3.2.2) and therefore it is guaranteed that the final configuration of each epoch snapshot is valid. □

3.4.5.3 Epoch Completeness

Epoch-completeness is satisfied via the the special initiation of the protocol based on epoch events as well as the alignment mechanism. In fact, the action of an epoch change at a source task corresponds to a marker-based snapshot initiation. Then alignment makes sure that all pending productions of an epoch, relevant to a task, are being processed before any message that belongs to a succeeding epoch.

Proof. There are three invariants associated with epoch-completeness that have to be satisfied by a snapshot $S_{\mathcal{C}_{ep_i}}$ of an epoch ep_i .

Inv. 1. $(\langle ep_i \rangle \prec e) \implies (e \notin \mathcal{C}_{ep_i})$

Inv. 2. $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$

Inv. 3. $((e \in \mathcal{C}_{ep_i}) \wedge (e \rightsquigarrow e')) \implies (e' \in \mathcal{C}_{ep_i})$

Proof [Inv. 2]: Given that epoch change events occur solely on source tasks, and no event at a source task is causally dependent on events occurring on other processes, the condition $e \prec \langle ep|i \rangle$ applies only to local event order at a source. Therefore, we only need to prove that for each source task $p \in \mathbb{A}$, $(e \leq_p \langle ep|i \rangle) \implies (e \leq_p e_p^{\odot})$. Based on the source task logic described in algorithm 5, the snapshotting action $e_p^{\odot}$ occurs at the reception of an epoch marker and thus, $e_p^{\odot} = \langle ep|i \rangle$. It therefore derives directly that invariant 2 is always satisfied. Note that the same epoch change occurs on Loop Head source tasks. The only difference in the case of Loop Heads is the in-transit logging of messages in a loop which correspond to pending productions of the currently snapshotting epoch (until the marker arrives back via the phantom channel), thus it is guaranteed that $(e \prec \langle ep_i \rangle) \implies (e \in \mathcal{C}_{ep_i})$ even in that case.

Proof [Inv.1+3]: Inv.1 refers to the exclusion of events corresponding to messages after an epoch change, while, Invariant 3 refers to the inclusion of all events caused by messages preceding an epoch change. We need to prove that both are satisfied on every task in the system. Given the algorithm variants we further separate the cases of source and regular tasks.

I) Source Tasks$(p \in \mathbb{A})$: Given the strict local execution order and the fact that $e_p^{\odot} = \langle ep|i \rangle$, inv.1 is always satisfied. Furthermore, Inv.3 considers event productions and hence it can never be violated for source tasks given that they don't consume input messages.

II) Regular Tasks$(p \in \Pi/\mathbb{A})$: The epoch alignment mechanism of algorithm 6 with FIFO channels suffice to satisfy both. According to epoch alignment, an input channel c_{pq} of a task $q \in \Pi$ becomes disabled from the time a marker arrives in a channel until the last marker is received and the process p executes the snapshotting and marker-forwarding action. For brevity, we will refer to those two events as $e_q^{\odot pq}$ and $e_q^{\odot}$ respectively and summarize the alignment condition in 3.11.

Alignment Condition: Given adjacent processes $p, q \in \Pi/\mathbb{A}$ the alignment mechanism enforces the following condition:

$$\nexists e_q = \langle proc, m_{pq}, M\rangle_q \in E_q : e_q^{\odot pq} \leq_q e_q \leq_q e_q^{\odot} \tag{3.11}$$

Via (3.11) the proof of Invariants 1 and 3 follows a similar logic: For Inv.1, assume by contradiction that $\langle ep|i\rangle \prec e_q \wedge e_q \in \mathcal{C}_{ep i_i}$. Based on the marker forwarding logic in a channel, let that be c_{pq}, it is known that $e_q^{\odot pq} \prec e_q$, since $e_q = \langle proc, m_{pq}, M\rangle$ (1). From (1) and (3.11) it can only be true that $e_q^{\odot pq} \leq_q e_q^{\odot} \leq_q e_q$ and thus, $e_q \notin \mathcal{C}_{ep_i}$ which leads to a contradiction $\Rightarrow\!\Leftarrow$.

Similarly, for invariant 3, given an event $e_p \in E_p$ and the production $e_p \rightsquigarrow e_q$, let us assume by contradiction the following: $(e_p \in \mathcal{C}_{ep_i}) \wedge (e_q \notin \mathcal{C}_{ep_i})$. Since $e_p \in \mathcal{C}_{ep_i}$, that event precedes the snapshotting event on process p: $e_p \leq_p e_p^{\odot}$ (2). Both of these events create productions in process q: $e_p \rightsquigarrow e_q$ and $e_p^{\odot} \rightsquigarrow e_q^{\odot pq}$. Via FIFO delivery (FIFORC4) we can maintain the delivery order of (2) to their productions from p to q given the fifo property of c_{pq}. Therefore, $e_q \leq_q e_q^{\odot pq}$ (3). However by applying the epoch alignment condition (3.11) to (3) we have $e_q \leq_q e_q^{\odot}$ and finally arrive to contradiction $e_q \in \mathcal{C}_{ep_i} \wedge e_q \notin \mathcal{C}_{ep_i} \Rightarrow\!\Leftarrow$.

We thus know that Inv.3 is satisfied for a single production. Via the transitive property of productions the proof extends trivially to arbitrary production trees via induction on the path of channels from p to q where $p \sim q$.

□

3.5 Summary

In this chapter, we introduced the fundamental execution model primitives and specifications of reliable data stream processing. Our analysis addresses a set of universal challenges when it comes to reliable stream processing, including the ability to recover from failures, reconfigure a continuous stream processing execution as well as to version and identify side effects of a distributed, event-based stream computation in a causally consistent manner. To that end, we proposed the Epoch-Based Stream Processing model according to which a stream computation is divided into a series of epochs that atomically commit the state of an execution. Furthermore, we showed how the epoch commit protocol can operate asynchronously to the stream computation via epoch-based snapshotting, a stricter form of a causally consistent snapshot that extracts a valid system configuration while respecting epoch order, a necessary property in our execution model. Our proposed marker-based mechanism can work on any weakly connected, static graph of processes with optional cycles and it has been proven to satisfy all necessary safety and liveness properties of epoch-based snapshotting. Given that this chapter focuses solely on the fundamentals, we skip the discussion on design principles and instead summarize the complete analysis in chapter 4. The integration of asynchronous epoch commits in Apache Flink further showcases how this underlying execution model can support complex operations in a widely deployed, production-grade scalable stream processing system.

CHAPTER

4

State Management in Flink

As presented previously in chapter 3, asynchronous epoch commits via snapshotting suffice to offer transactional processing guarantees without blocking synchronization and coordination in the context of distributed data streaming. This chapter presents in more detail a concrete adaptation of these techniques within Apache Flink's end-to-end runtime. We describe how the principles we introduced work in practice and drive stateful processing inside and outside the Flink system. Flink's runtime can support continuous long-running task executions without interruptions, while allowing side effects of epochs to commit asynchronously, outside the critical path of the computation. Aside the core snapshotting mechanism, many other related operational mechanisms are made transparent to the user, posing no restrictions on Flink's expressive programming model and thus allowing arbitrary operations on partitioned state.

The outline of the chapter goes as follows: section 4.1 describes the internal runtime components of Flink that participate in the state aquisition mechanism and the concrete end-to-end epoch commit protocol as well as its interactions with various backends developed by the Flink community. Then, section 4.2 [1]offers an in-depth overview of Flink's reconfiguration mechanism and related choices regarding repartitionable state in the system. Section 4.3 introduces the concept of external query isolation and application execution provenance tracking, two useful operations that build on epoch commits. Then, section 4.4 presents the costs and benefits of epoch-based snapshots based on production statistics gathered during a long large-scale deployment of Flink. Finally, section 4.5 discusses existing and potential optimisations such as incremental snapshots and dynamic reconfiguration, followed by acknowledgements in section 4.7 and a summary in section 4.8 covering the use of our core design principles.

[1]Sections 4.2 and 4.4 include previously-published content [29] as-is.

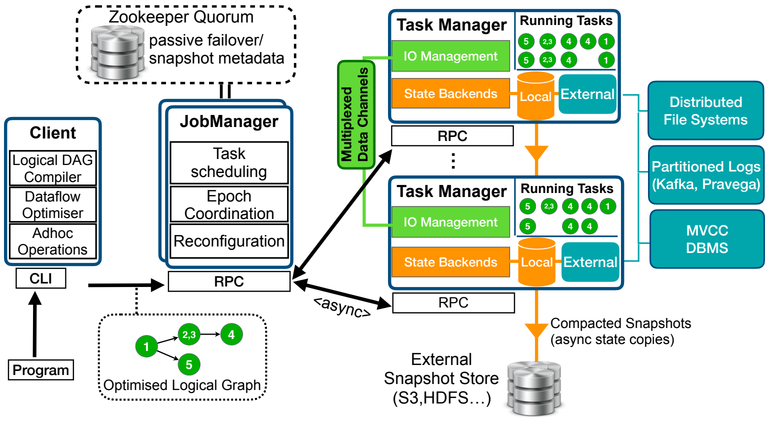


Figure 4.1: A Detailed Overview of Flink's Runtime.

4.1 Asynchronous Epoch Commit Integration

Flink's distributed runtime has a master-slave architecture similar to Spark and Hadoop MapReduce. Yet, in contrast to batch-centric application management [12] that builds on staged, short-running computation, Flink employs a schedule-once, long-running allocation of tasks. Nevertheless, through the use of epochs and corresponding state backends that abstract operations on state the system is able reconfigure applications (e.g., scale-out) and re-allocate application state on-demand while offering reliable processing guarantees (via atomic epoch commits). This approach minimizes management overhead while allowing for further adaptation to hardware or software changes or partial failures that can potentially occur. We first explain the overall design choices and then focus on each respective mechanism related to epoch-based snapshots in more detail.

4.1.1 Architecture Breakdown

In Figure 4.1 we provide an overview of all Flink runtime components and their purpose, which we describe in more detail below.

Client: The client library provides support for static type checking and compilation of Flink programs. In chapter 2 we sketched most operational semantics of stream operations from the programmer's perspective. An operation corresponds to a unary (e.g., map, filter, fold, window) or n-ary higher-order function (e.g., join, co-map, co-flatmap) and is parametrized with user-defined function literals. The Flink client compiles operations to a logical graph of tasks, encapsulating each independent application component and its data dependencies to other components.

A task in the final optimised graph corresponds to a single or multiple (chained) logical operators compiled by Flink's graph generator and optimiser (see chapter 2). The client provides a command line interface (CLI) to compile, submit, monitor and reconfigure running applications and thus serves as a complete interface to Flink's distributed runtime.

JobManager: The JobManager is a JVM process that maintains a global view and control of each running application, including corresponding tasks and their locally managed states. The role of the JobManager is crucial to epoch-based execution since it is the intermediary that triggers epoch-change events, collects recorded snapshots and notifies back tasks about completed epochs, as part of the asynchronous epoch commit protocol. Furthermore, it employs heartbeat-based failure-detection and monitoring of all Flink cluster resources. Every action of the Job Manager that alters the metadata of an application (e.g., scheduling, reconfiguration, epoch completion etc.) is first committed in a Zookeeper quorum [77]. This allows for passive-standby deployments that can guarantee a consistent system execution which can deal with all types of failures, including master node failures. All communication with the client and the local workers (TaskManagers) respects an asynchronous RPC-based protocol that does not interfere with data channels used by the application tasks.

TaskManager: Each logical task of an application is scheduled and physically executed in parallel across multiple workers. Physical tasks are granted access to two major resources: network channels and state. The overall resources available in a worker should be shared efficiently and in isolation by multiple physical tasks (typically assigned to dedicated containers using YARN [48] or Mesos [49]). This is the work of the TaskManager JVM process, which also serves as the main proxy between physical tasks and the JobManager. JobManager processes are stateless and employ policy-based networking and state access while executing requests received from the JobManager. Data channels between task threads are multiplexed into shared TCP connections with shared buffers for serialization/deserialization needs and deadlocks are typically avoided through in-flight flow control. State backends are a modular way to make state operations transparent to the physical state's location and representation. Flink supports plug-in backends for locally embedded databases (e.g., RocksDB), external partitioned logs (Kafka, Pravega, Kinesis etc.) and external multiversion concurrency control-enabled (mvcc) databases without requiring any changes in the user program. Most importantly, physical state is almost always kept outside the heap and managed externally to the JVM for scalability and performance (no garbage collection).

4.1.2 Task Design and Process Model

Physical tasks in Flink adopt the stream process model presented in subsubsection 3.3.1.1 and therefore follow a strict message-based control flow, manipulating

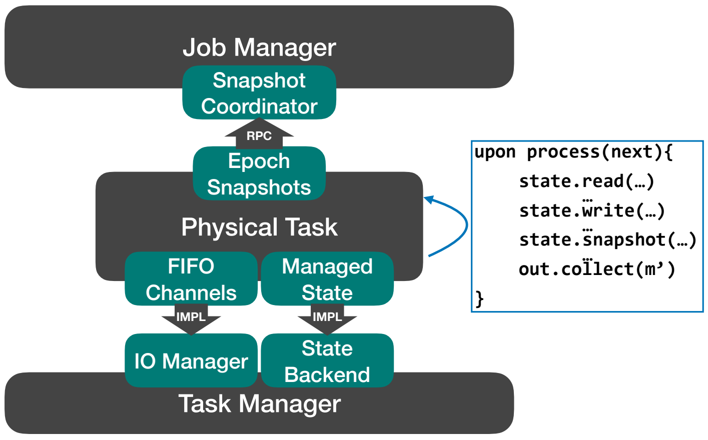


Figure 4.2: Component Design of Physical Tasks.

state and generating output records within an atomic action triggered by an input message. In an epoch-based execution with reliable processing guarantees no indirect communication or external connections are supported in that critical path. However, external asynchronous communication mechanisms are used for auxiliary purposes such as the epoch commit protocol presented later.

In Figure 4.2 we depict how physical tasks are modeled from a component design point of view. The logic within a physical task gets invoked per input message received from one of its FIFO channels. A task can read, write or issue snapshotting operations on its managed state as well as send output messages through a collector interface ("collect" flushes messages downstream). Both implementations of FIFO channels and managed state are provided by the TaskManager. Furthermore, at the end of a complete epoch, each task provides a reference of its snapshotted state which is collected via asynchronous RPC issued by the Job Manager. For convenience, the channel prioritization logic required by the alignment phase of the epoch-based protocol is implemented by the FIFO channels provided by the IOManager component.

4.1.3 Protocol Implementation

Flink makes use of the Asynchronous Epoch Commit protocol (see chapter 3 section 3.3) , guaranteeing that all events in an execution up to an epoch and their internal and external state operations are atomically committed to stable storage. The implemented protocol includes all asynchronous communication steps between a Snapshot Coordinator process (JobManager), Physical Tasks (TaskManager) and respective state backends. An instance of the protocol runs per epoch change and is initiated by the coordinator while supporting concurrent instances of the protocol.

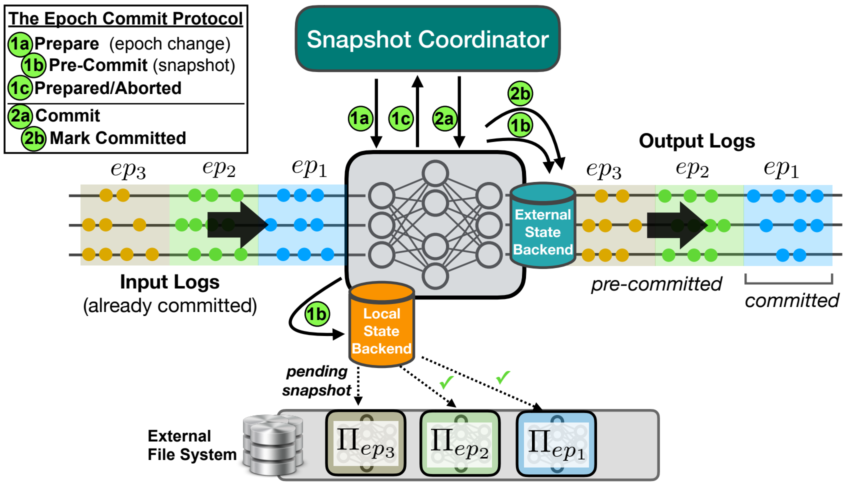


Figure 4.3: Overview of Epoch Commit with Snapshots.

If an instance gets aborted (e.g., due to a partial failure) the system rolls-back all pre-committed changes and the execution restarts from the latest committed epoch. Most importantly, the epoch commit protocol is being executed concurrently with the physical task execution and does not influence the critical part of the computation. In Figure 4.3 we visualize each communication step of the epoch commit protocol and further explain each step below.

Prepare Phase: The prepare phase starts once the snapshot coordinator issues an epoch change. This can occur periodically (e.g., every 20 seconds) or as an adhoc request from the user. In both cases, the coordinator broadcasts an epoch change message (1a) to all TaskManager nodes which in turn initiates the epoch-based snapshotting protocol described in section 3.4 at all source tasks including marker dissemination and epoch alignment. Eventually, if no failures occur, every physical task triggers a local snapshot on its managed state and an acknowledgment is sent back to the coordinator (record notification). We call each local snapshotting operation a pre-commit step (1b). In the case of operators with a local state backend the pre-commit is a copy operation of the state to an external file system. However, in the case of an external state backend, a pre-commit locks remote changes to allow for no further operations and to prepare it for the final commit. Nevertheless, neither the snapshotted states nor the externally pre-committed states should be accessible at this point since the epoch is not yet committed. The prepare phase ends once all tasks have notified the coordinator with a “prepared” acknowledgment (1c). That can only happen after the snapshotting algorithm has finished. If a partial error (abort message) or global timeout occurs during the process the protocol aborts.

Commit Phase: The commit phase begins once all "Prepared" commands are received by the coordinator. The purpose of the commit phase is to confirm all pre-commited states for external access. The Snapshot Coordinator initiates the commit phase by broadcasting a "Commit Epoch" message to all tasks (2a) which in turn commit pending epochs at the backends (2b). This action can be invoked concurrently to the task execution by the TaskManager, thus, inducing no performance impact on the critical execution path. The commit phase is especially important for external state backends in order to make all external changes visible to the outside (e.g., pre-committed output streams). Failures can potentially occur during this phase which can lead to incomplete committed changes in an epoch. However, this does not violate any system guarantees. Given that all changes are at least guaranteed to be pre-committed at this point, pending commits can be eventually re-issued during the rollback mechanism (explained below) to finalize the process.

Summary: Epoch commit is a special-purpose two-phase commit protocol that provides transactional processing guarantees. The usage of asynchronous epoch-based snapshots for pre-committing all side-effects yields several important observations. First, no commit operation affects the runtime performance (i.e., throughput) of the system. The commit protocol simply makes all side effects of an epoch externally visible once it is guaranteed that everything is stored to some form of stable storage. Second, it allows pipelining of multiple concurrent epoch commit instances with the use of asynchronous epoch-based snapshots. In the example of Figure 4.3, we can see the case of three possibly concurrent epochs. For epoch ep_1 all side effects are committed while ep_2 has been pre-committed and the commit phase is pending. Finally, ep_3 is in the pre-commit phase which means that the snapshotting algorithms is being currently executed. In the same example, it is safe to rollback from ep_2 given that all states are stored to stable storage at that instant.

4.1.3.1 The Rollback Procedure

Flink's rollback mechanism is initiated either when a partial failure gets detected during normal operation (fail-stop model), or when reconfiguration is requested or upon an aborted epoch commit instance. Rollback respects a "stop, reschedule and restore" procedure whether the reason is failure recovery or reconfiguration (see Figure 4.4). In all cases, the system (JobManager) picks the latest prepared (but not necessarily committed) snapshot to restart an execution from. Eventually, all states within the pipeline are progressively retrieved and applied to reflect an exact, valid distributed execution at the restored epoch. Below, we identify several special cases of a task rollback:

Loop Head Tasks: In the case of Loop Head tasks, all records logged during

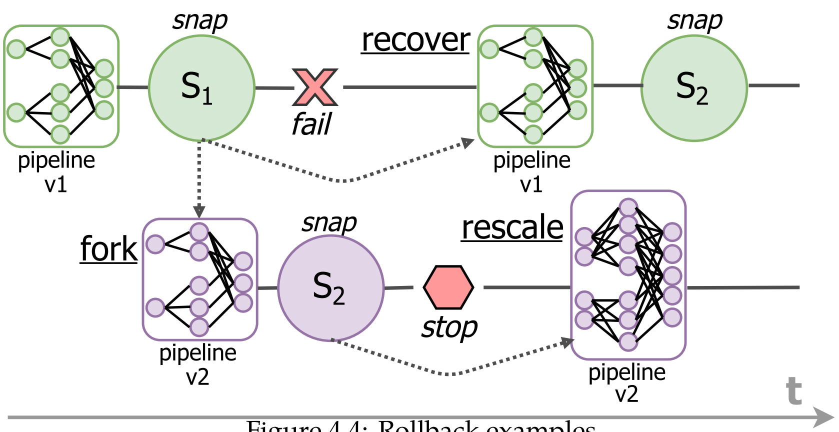


Figure 4.4: Rollback examples.

the snapshot are recovered and flushed to output channels prior to their regular record-forwarding logic. This way, it is guaranteed that the state of the loop returns back to its snapshotted execution (that is, the sum of the in-transit records).

State in External Backends: In order to circumvent the case of permanently uncommitted external states, all external state backends issue a preemptive commit in the beginning of a rollback in order to guarantee that no uncommitted changes will persist past the recovered execution point.

Regular Sources: All data sources need to restore their (deterministic) execution back to the current offset of each stream when the snapshot occurred. Flink's data sources provide this functionality out-of-the-box by maintaining offsets to the latest record processed prior to an epoch from external logging systems. Upon recovery the aggregate state of those sources reflects the exact distributed ingestion progress made prior to the recovered epoch. This approach assumes that external logging systems, that sources communicate with, index and sequence data across partitions in a durable manner (e.g., Kafka, Kinesis, PubSub and Distributed File Systems).

Selective Rollback: Depending on the rollback cause, certain optimized recovery schemes can be employed. For example, during a full restart or rescaling, all tasks are being redeployed, while after a failure only the tasks belonging to the affected connected component (of the execution graph) are reconfigured, if more than one connected components exist.

In essence, known incremental recovery techniques from micro-batch processing [12] are orthogonal to Flink's rollback approach and can also be employed. A snapshot epoch acts as synchronization point, similarly to a micro-batch or an input-split. On recovery, new task instances are being scheduled and, upon initialization, retrieve their allocated shared of state back to their respective backends, from

	Start	**Pre-Commit**	**Commit**	**Abort**
Local (RocksdB)	new MemTable	flush MemTable / Snapshot SSTAbles	-	delete snapshot
Local (Heap)	-	deep copy (full snapshot)	-	delete snapshot
Kafka $\geq$0.11	new transaction	close/new transaction	commit transaction	abort transaction
Pravega	new Segment	seal Segment	commit Segment	delete Segment
HDFS	create File in tmp dir	close File (no writes)	move (atomic) file to committed Dir	truncate/delete file
DBMS (non-MVCC)	new WAL	snapshot WAL	execute WAL as transaction	drop WAL
DBMS (MVCC)	-	new version	incr version	decr version

Table 4.1: Examples of Backend-Native Operation Mappings to Flink's Epoch Commit Protocol

externally stored snapshots.

4.1.4 Backend Integration

Several backend implementations have been contributed to Flink by its developer community which integrate seamlessly with Flink Epoch Commit protocol. In Table 4.1 we summarize some of the existing and possible backends and the required operations to integrate with Flink's Epoch Commit. In general, each backend supports four operations: 1) Start, 2) Pre-Commit, 3) Commit and 4) Abort. For several backends that allow some form of transactional writes the operational needs are satisfied out of the box. These include recent versions of partitioned logs such as Kafka (≥ 0.11) and Pravega as well as DBMSs with multi-version concurrency control. For example Apache Kafka 0.11 introduced support for exactly-once delivery for producers. Flink's Kafka 0.11 transactional sink basically issues a new transaction per epoch which can be committed atomically by Kafka itself. Pravega is another example of a newer system which comes with built-in support for transactional cross-partition writes already in its model. Pravega's segments represent self-contained partitioned logs which can be atomically started and sealed, integrating tightly with Flink's epoch-based processing since a segment per epoch can be started, sealed (committed) and deleted in case of an epoch abort.

In the rest of the cases, the implementation of two-phase commit has to be solved indirectly by the developer of the backend. Examples of non-transactional backends are non-MVCC databases and File Systems. For example, the HDFS transactional sinks in Apache Flink persist all state append operations to temporary files and rely on HDFS truncate to abort an epoch and atomic move operations to move a closed HDFS file of an epoch to a "read-committed" directory. In the case of DBMSs or other external store systems a general strategy that works at an additional latency and storage cost is to maintain a Write-Ahead-Log (WAL) for all uncommitted external operations in local state. Upon an epoch commit phase (or recovery) the WAL can be executed and get discarded from the local state.

In general local backends are needed to complement the snapshotting of all metadata needed to persist external two-phase commits. For example, WALs, log offsets, transaction IDs etc. are important metadata that requires bookkeeping to eventually finalize any pending asynchronous message-exchange protocols that have been initiated with external systems.

4.2 System Reconfiguration

Asynchronous epoch snapshots and the rollback procedure cover the needs of reconfiguration but only in part. A typical need in any data-intensive application deployment is to be able to modify the scale (i.e., parallelism) of certain logical tasks. For tasks that have declared managed state we need to consistently allocate data

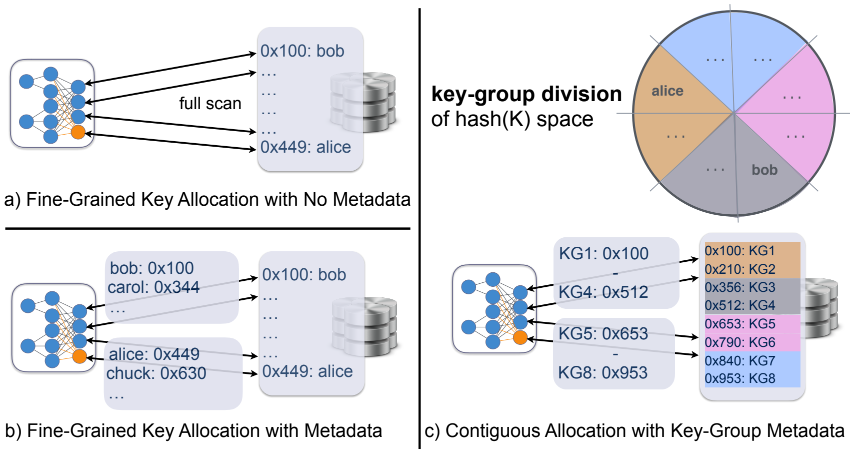


Figure 4.5: State Allocation and Metadata Alternatives

stream partitions and further re-allocate in the case of reconfiguration. In chapter 2 we covered the different managed state representations in Apache Flink. In fact, most scalable operations and respective state are scoped by a user-defined key with a key space K. For load balancing, both streams and respective states are sharded in the space of a consistent hashing function $h : K \rightarrow \mathbb{N}^+$.

4.2.1 Key-Group Partitioning and Allocation

Flink decouples key-space partitioning and state allocation similarly to Dynamo[78]. The runtime maps keys to an intermediate circular hash space of "key-groups" : $K^* \subset \mathbb{N}^+$ given a maximum parallelism π-max and a hash function h as such:
$K^* = \{h(k) \bmod \pi\text{-max} \mid k \in K, \pi\text{-max} \in \mathbb{N}^+, h : K \rightarrow \mathbb{N}^+\}$

Given that snapshots should contain all information needed to find and re-allocate state, there is an evident trade off between the overhead of a rollback (I/O during state scans) and snapshot metadata needed to re-allocate state to different numbers of instances. On one extreme each parallel task could scan the whole state (often remotely) to retrieve the values of all keys assigned to it. This yields significant amounts of unnecessary I/O (Figure 4.5(a)). On the opposite extreme, snapshots could contain references to every single key-value and each task could selectively access its assigned keyed states (Figure 4.5(b)). However, this approach increases indexing costs (proportional to num. of distinct keys) and communication overhead for multiple remote state reads, thus, not benefiting by coarse-grained state reads. Key-groups (Figure 4.5(c)) offer a substantial compromise: reads are limited to data that is required and key-groups are typically large enough for coarse grained reading (if π-max is set appropriately low). In the uncommon case where

$|K| < \pi$-max it is possible that some task instances simply receive no state. Finally, this mapping ensures that a single parallel physical task will handle all states within each assigned group, making a key-group the atomic unit for re-allocation.

4.2.2 State Re-Allocation

To re-assign state, we employ an equal-sized key-group range allocation. For π parallel instances, each instance $t^i \in \Pi, 0 \leq i \leq \pi$ receives a range of key-groups from $\lceil i \cdot \frac{\pi\text{-max}}{\pi} \rceil$ to $\lfloor (i+1) \cdot \frac{\pi\text{-max}}{\pi} \rfloor$. Seeks are costly, especially in distributed file systems. Nevertheless, by assigning contiguous key-groups we eliminate unnecessary seeks and read congestion, yielding low latency upon re-allocation. `Operator-State` entries, which cannot be scoped by a key, are persisted sequentially (combining potential finer-grained atomic states defined across tasks), per operator, within snapshots and re-assigned based on their *redistribution pattern*, e.g., in round-robin or by broadcasting the union of all state entries to all operator instances.

4.3 Operations with Epoch Snapshots

Fault tolerance and reconfiguration are only a subset of the potential needs and benefits covered by epoch-based snapshotting. In this section, we introduce another two novel use-case examples of snapshots, namely external access isolation guarantees for managed state and application provenance both of which have been examined and prototyped in Apache Flink.

4.3.1 External Access Isolation

Flink allows direct adhoc queries to its managed state from outside the system. This way external systems or users can access Flink's keyed-state in a similar way as that of a key/value store, providing read-only access to the latest values computed by the stream processor. This feature is motivated by two observations. First, it is required by many applications to grant ad-hoc access to the application state for faster insights. Secondly, eager publishing of state to external systems frequently becomes a bottleneck in the application as remote writes to the external systems cannot keep up with the performance of Flink's local state on high-throughput streams.

Queryable state can be accessed via a subscription-based API. First, managed state that allows for query access is declared in the original application. Upon state declaration (see chapter 2 section 2.1) it is possible to allow access from external queries by simply setting a flag in the descriptor that is used to create the actual state, having an assigned unique name for this specific state to be accessed, as such:

```
//stream processing application logic
val descriptor: ValueStateDescriptor[MySchema] = ...
```

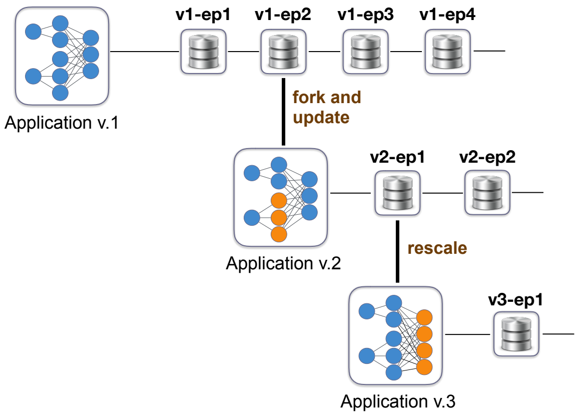


Figure 4.6: Application Provenance with Snapshots

```
descriptor.setQueryable("myKV")
...
val mutState: ValueState[MySchema] = ctx.getState(descriptor)
```

Upon deployment, a state registry service gets initiated and runs concurrently with the task that holds write access to that state. A client that wishes to read the state for a specific key can, at any time, submit an asynchronous query (obtaining a `future`) to that service, specifying the job id, registered state name and key, as shown below:

```
//client logic
val client = QueryableStateClient(cfg);
var readState: Future[_] = client.getKVState(job, "myKV", key);
```

The current implementation of queryable state supports point lookups of values by key. The query client asks the Flink master (JobManager) for the location of the operator instance holding the state partition for the queried key. The client then sends a request to the respective TaskManager, which retrieves the value that is currently held for that key from the state backend.

From a database query isolation-level viewpoint, such queries access *uncommitted state*, thus following the *read-uncommitted* isolation level. However, via the use of snapshots it is possible to offer *read-committed* isolation support by letting TaskManagers hold onto the state of committed snapshots, and use that state to execute adhoc queries.

4.3.2 Application Provenance and Migration

Epoch-based executions make application provenance possible and more importantly, trivial. This is due to the fact that snapshots mark a clear dependency chain between epochs and reconfiguration actions applied throughout the history of a long-running stream processing application. As shown in Figure 4.6, an application execution dependency diagram resembles that of a version control system (e.g., git), where its distinct change is encapsulated into epoch snapshots. Furthermore, an application can have multiple versions, forked from existing snapshots that are being executed in different configurations (e.g., different parallelism, cluster, logic etc.). This can further ease the development and maintenance of continuous applications as well as granting the freedom to issue bug fixes that can rollback in the past (e.g., at an old epoch snapshots) and thus, re-conciliate erroneous functionality starting from the time it actually occurred.

Another important aspect of snapshots is that they make a continuous execution of an application purely portable. Migrating a full pipeline is as simple as rolling back the application to a committed epoch. Given that snapshots themselves are blobs that can be moved to different locations, this grants an important flexibility to application developers since it makes an entire transition to different cloud providers or on-premise clusters trivial.

4.4 Performance Analysis

At the time of writing, Flink has gone beyond a research prototype and is one of the most widespread open source systems for data stream processing, serving the data processing needs of companies ranging from small startups to large enterprises (e.g., Uber, Netflix, Alibaba, Hwawei, Ericsson, King, Zalando etc.). In the remainder of this section, we present a performance analysis derived from live production metrics heavily focused on the performance costs and benefits of asynchronous epoch-based snapshots, which is the core contribution of this work. The data used in this analysis has been extracted from production server logs at King (King Digital Entertainment Limited), a leading mobile gaming provider with over 350 million monthly active users.

4.4.1 A Real-Time Analytics Platform

The Rule-Based Event Aggregator (RBEA) by King [79], is a reliable live service that is implemented on Apache Flink and used daily by data analysts and developers across the company. RBEA showcases how Flink's stateful processing capabilities can be exploited to build a highly dynamic service that allows analysts to declare and run standing queries on large-scale mobile event streams. In essence, the service covers several fundamental needs of data analysts: 1) instant access to

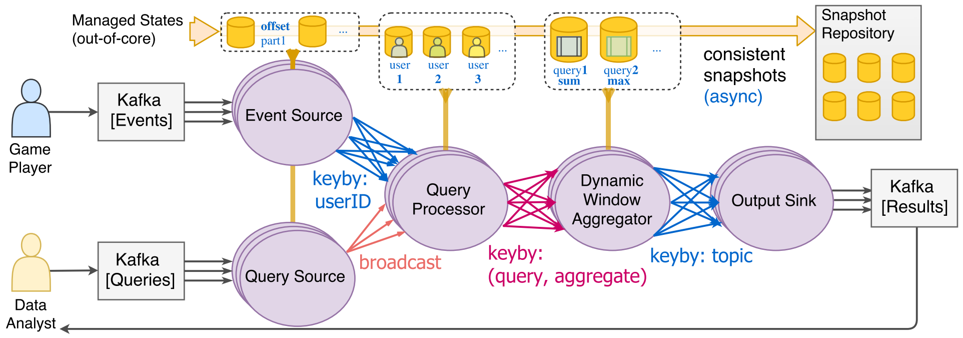


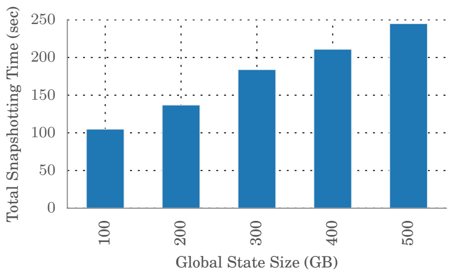
Figure 4.7: Overview of the Flink pipeline implementing an adhoc standing query execution service at King

timely user data, 2) the ability to deploy declarative standing queries, 3) creation and manipulation of custom aggregation metrics, 4) a transparent, highly available, consistent execution, eliminating the need for technical expertise.

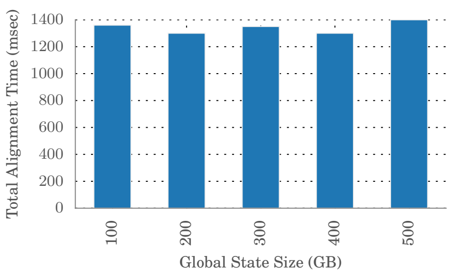
4.4.1.1 The RBEA Service Pipeline

Figure 4.7 depicts a simplified overview of the end-to-end Flink pipeline that implements the core of the service. There are two types of streams, ingested from Kafka: a) an `Event` stream originating from user actions in the games (over 30 billion events per day) such as `game_start/game_end` and b) a `Query` stream containing standing queries in the form of serialized scripts written by data analysts through RBEA's frontend in a provided DSL (using Groovy or Java). Standing queries in RBEA allow analysts to access user-specific data and event sequences as well as triggering special aggregation logic on sliding data windows.

Standing queries are forwarded and executed inside `[Query Processor]` instances which hold managed state entries per user accumulated by any stateful processing logic. A "broadcast" data dependency is being used to submit each query to all instances of the `[Query Processor]` so it can be executed in parallel while game events are otherwise partitioned by their associated user ids to the same operator. Aggregation calls in RBEA's standing query DSL trigger output events from `[Query Processor]` operator which are subsequently consumed by the `[Dynamic Window Aggregator]`. This operator assigns the aggregator events to the current event-time window and also applies the actual aggregation logic. Aggregated values are sent to the `[Output sink]` operator which writes them directly to an external database or Kafka. Some details of the pipeline such as simple stateless filter or projection operators have been omitted to aid understanding as they don't affect state management.



(a) Epoch Prepare Duration vs Total Size [π:70, `state`:[100:500GB], `hosts`:18]



(b) Alignment Time vs Snapshot Size [π:70, `state`:[100:500GB], `hosts`:18]

Figure 4.8: RBEA Deployment Measurements on Snapshots

4.4.1.2 Performance Metrics and Insights

The performance metrics presented here were gathered from live deployments of RBEA over weeks of its runtime in order to present insights and discuss the performance costs related to snapshotting, as well as the factors that can affect those costs in a production setting. The production jobs share resources on a YARN cluster with 18 physical machines with identical specification each having 32 CPU cores, 378 GB RAM with both SSD and HDD. All deployments of RBEA are currently using Flink (v.1.2.0) with local out-of-core RocksDB state backend (on SSD) which enables asynchronous local snapshot invocation for copying the full application state to HDFS (see section section 4.5 on asynchronous snapshot invocation). The performance of Flink's state management layer, that we discuss below, has been evaluated to address two main questions: 1) What affects snapshotting latency?, and 2) How and when is normal execution impacted?

1) What affects snapshotting latency?
We extracted measurements from five different RBEA deployments with fixed parallelism $\pi = 70$ ranging from 100 to 500 GB of global state respectively (each processing data from a specific mobile game). Figure 4.8(a) depicts the overall time it takes to undertake a full epoch prepare phase (full snapshotting time) asynchronously for different state sizes. Mind that this simply measures the time difference between the invocation of a snapshot (begin of Prepare phase) and the moment all operators notify back they have completed it through the asynchronous backend calls (Prepared). As snapshots are asynchronously committed these latencies are not translated into execution impact costs, which makes alignment the sole factor of the snapshotting process that can affect runtime performance (through partial input blocking). Figure 4.8(b) shows the overall time RBEA task instances have spent in *alignment* mode, inducing an average delay of 1.3 seconds per full snapshot across all deployments. As expected, there are no indications

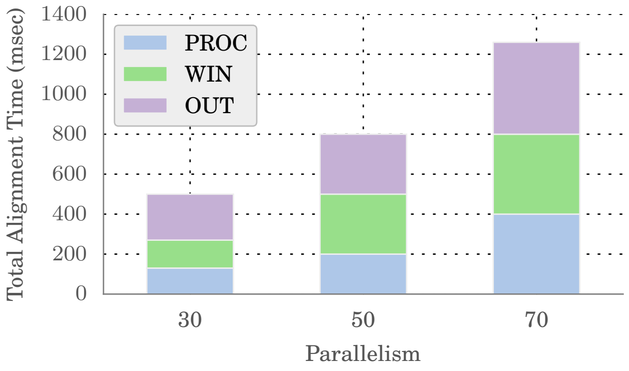


Figure 4.9: Alignment Time vs Parallelism [π : [30 : 70], `state`:200GB, `hosts`:18]

that alignment times can be affected by the global state size. Given that state is asynchronously snapshotted, normal execution is also not affected by how much state is snapshotted.

2) How and when is normal execution impacted?
Alignment employs partial blocking on input channels of tasks and thus, more connections can introduce higher runtime latency overhead. Figure 4.9 shows the total times spent aligning per full snapshot in different RBEA deployments of fixed size (200GB) having varying parallelism. Evidently, the number of parallel subtasks π affects the alignment time. More concretely, the overall alignment time is proportional to two factors: 1) the number of shuffles chained across the pipeline (i.e., RBEA has 3× `keyby` for the `PROCESSOR`, `WINDOW` and `OUTPUT` operators respectively), each of which introduces a form of alignment "stage" and 2) the parallelism of the tasks. Nevertheless, occasional latencies of such a low magnitude (~1sec) are hardly considered to be disruptive or breaking SLAs, especially in highly utilized clusters of such large-scale deployments where network spikes and CPU load can often cause more severe disruptions.

4.5 Additional Notes and Optimisations

It should be noted that certain performance metrics such as rollback recovery times, the use of incremental snapshots and more are not included in our analysis since they are orthogonal to asynchronous epoch snapshots and vary heavily between different state backends. At the core of asynchronous epoch-based snapshotting there is only a single "cost" within the critical path of an application, that of alignment. Epoch commit latencies depend on how each respective backend commits changes as well as which local backend is used (which affects the speed at which an epoch is pre-committed). In this section we address several important state management optimisations that have been implemented by the Flink community to boost the system's performance, as well as known optimisations that can be incorporated in the future.

4.5.1 Asynchronous Snapshot Invocation

The asynchronous epoch-based snapshotting protocol (algorithm 6) employed by Flink obtains a recorded state as a local copy. There is yet another important level of asynchrony, orthogonal to the protocol, that of the invocation of the state copy. In reality, most state backends of Flink can execute the local snapshotting operation asynchronously. The snapshotting protocol can still accurately pinpoint the exact instant on which the snapshot can, otherwise asynchronously, be obtained. Flink's core backends such as RocksDB provide native support for asynchronous snapshots. In RocksDB [47] a snapshot operation flushes the memTable (uncommitted operation log) to disk, creating an SSTable (persisted operation log) and a copy process to an external file directory begins concurrently. Once copying is complete the notification logic to the snapshot coordinator triggers back asynchronously without going through the critical path of the application. Asynchronous snapshots are especially important since any synchronous copies can introduce significant latencies to the application, especially when full snapshots are employed. In the performance analysis presented in section 4.4 we only considered snapshots with asynchronous invocation. This allowed us to focus on the actual cost of the protocol (epoch alignment) while eliminating all backend-specific performance concerns.

4.5.2 Incremental Snapshots

So far it might have seemed counterintuitive to copy a full global state of an application per epoch. States can be as big as many Terabytes of data, generated by billions of events throughout an application's long lifetime. While the existence of a full state is important for many of the purposes presented before (e.g., state queries and reconfiguration) it is still possible to replace full snapshots with incremental snapshots. Incremental snapshots encapsulate only the changes applied to the managed state (i.e., writes, appends) between epochs. The actual implementation and usage of incremental snapshots relies on the backend that implements it. For example, Flink enables incremental snapshots through its RocksDB backend. In fact, copying only the "commit log" of operations on state is a natural procedure in RocksDB due to the fact that it already keeps all operations organized hierarchically in commit logs that are backed to disk. Since a full application state is needed periodically for a correct rollback, the system periodically compacts incremental snapshots into a full snapshot, thus, allowing applications to recover from specified epochs. It has been reported that incremental snapshotting can offer orders of magnitude of speedup, such as from 3 minutes to 30 seconds for applications with Terabytes of state [80]. Therefore, in addition to reduced storage requirements, this feature can further lead to lower epoch commit latency, making side effects accessible in constantly less time.

4.5.3 Omitting Epochs

It is common in production deployments to face errors during a copy of a large snapshot (e.g., due to a temporary HDFS, S3 error). That means that certain instances of the epoch commit protocol might be potentially aborted. To make an application execution less sensitive to temporary external faults, the system can tolerate up to a specified number aborted snapshots. Mind that, when it comes to applications without external side effects (i.e., sinks) an arbitrary number of epochs can be ignored. If, for example, we omit snapshot of ep_2 but successfully obtain a snapshot of ep_3 e.g., Π_{ep_3} we can still offer reliable processing guarantees by omitting Π_{ep_2} since $E^{ep_3} \subset E^{ep_3}$. In fact, for fault tolerance purposes the system holds only the snapshots of the latest committed epoch. In case where transactional sinks are used and we want to maintain reliable processing guarantees all epochs have to eventually be committed in external state backends. In that case, the epoch commit protocol should repeat up to a number of retries until completion.

4.5.4 Relaxed Guarantees

In several application cases, strongly transactional processing is not a strict requirement and users are satisfied with at-least once processing guarantees. Conceptually, at-least-once means that given a deterministic stream input, every event production should occur at least one time, thus duplicate productions are allowed. Flink can provide this type of relaxed guarantees with a minor modification to its snapshotting algorithm, simply by omitting channel prioritization during alignment. If in algorithm 6 we simply keep all channels in the pending list and snapshot, as before, once all input barriers of an epoch ep_n arrive we have a case of a snapshot E' for which $E^{ep_n} \subseteq E'$. If we rollback an application from that snapshot all input records and productions of epochs $\geq n$ will occur and thus, any additional events that were included in the snapshot will be repeated. Flink allows for at-least once processing guarantees as a configuration option which employs this strategy, further eliminating the sole cost of epoch-based snapshots, that of alignment.

4.5.5 Dynamic Reconfiguration

The epoch alignment mechanism makes sure that each physical task completes all work of the current epoch before going further with any processing into the next. For cases of reconfiguration that do not alter the underlying structure of the stream process graph but are limited to logical changes (e.g., bug fix in a user-defined function) it is possible to dynamically apply all changes without violating validity or epoch completeness across configurations. The main idea is to encapsulate the patch within the epoch markers and apply it at the instant that local check-pointing takes place. This way we can guarantee that conceptually all changes are applied after the completion of the epoch and no task will received records corresponding to the updated logic prior to an epoch completion, again due to alignment. This technique can potentially be integrated on Flink or other systems that use the same algorithm.

4.5.6 Idempotent Sinks

A special, simplified case in the epoch commit protocol is that of deterministic applications. This can be the case when no logical operator in the application is sensitive to order (e.g., restricted to progress-based operators such as event-time windows). In those cases each rollback will yield the exact same external (and internal) side effects on its output sinks. Repeated sink commits can therefore support idempotency (e.g., leading to the same key value store writes or database queries) and the epoch commit protocol can be ignored all together. That is because in that case we will always have the same external side effects no matter how many times all events occur internally due to a rollback.

4.5.7 Hybrid Fail-Stop and Recovery Model

One of the main assumptions made throughout this work has be then one of embedded *volatile* state and the fact that any data stored in the working memory of each task is lost upon a failure. The recent development and possible future adoption of more affordable *non-volatile random-access memory* (NVRAM) as well as Remote Direct Memory Access (RDMA) for compute clusters are some of the factors that can play a key role in the future for next-generation processing systems. These hardware advances would possibly make fine-grained fault-recovery [10, 11] more attractive for fault tolerance given that active state can always be accessed across distributed compute resources even upon process failures. Despite the beneficial usages in fault-recovery the need to address the full application state consistently and execute system-wide operations such as querying committed state or migrating logic would still require a form of a commit mechanism. In that context, our asynchronous epoch-commit protocol could be defacto employed to satisfy these needs using snapshots while allowing for local fine-grained recovery models to also be used seemlesly in par.

4.6 Related Work

Reliable Dataflow Processing: Flink offers coarse grained, job-level snapshot maintenance which grants various operational benefits. Flink bears many similarities to SEEP [11, 10] in terms of employing partitioned embedded state and a form of snapshotting for fault tolerance. Yet SEEP's main focus is not end-to-end transactional processing but rather fault tolerance, scalability and repartitionable state, thus, it has not adopted an application-wide commit mechanism. IBM Streams employs a pipelined checkpointing mechanism [64] that executes in-flight with data streams as with Flink's, tailored to weakly connected graphs with potential cycles. The most distinct difference to Flink's approach is that IBM Streams adopted a stricter snapshotting scheme: 1) First, all records in transit are consumed in order to make sure that they are reflected in the global state while blocking all outputs. 2) All operators trigger their snapshot in topological order, using markers as in our technique and resume normal operation. Flink's protocol only drains records within respective cycles. Furthermore, Flink's alignment is a local operation and does not halt global progress or hold up output in an execution graph making it more transparent and non-intrusive. Finally, IBM Streams supports language abstractions for selective fault tolerance. On Flink, the choice of snapshotting state is achieved by simply using managed state versus unregistered state, without requiring further user intervention. In the scope of a pipeline/component, snapshots can also be enabled or disabled through Flink's configuration.

Apache Storm [5] initially offered only guaranteed record processing through record dependency tracking. However, the most recent releases of Storm (and Apache Apex[39]) incorporated a variant of Flink's algorithm to its core in order to support transactional processing guarantees. Meteor Shower [81] employs a similar alignment phase to Flink. However, it cannot incorporate cyclic dataflow graphs which is a common case for online machine learning [82] and other applications. The same solution does not cover state rescaling and transparent programming model concerns. Naiad [20] and the sweeping checkpointing technique enforce in-transit state logging even in subgraphs where cycles are not present. Moreover, Naiad's proposed three phase commit disrupts the overall execution for the purpose of snapshotting. Finally, MillWheel [9] offers a complete end-to-end solution to processing guarantees, similarly to Flink. However, its heavy transactional nature, idempotency constraints and strong dependence on a high-throughput, always-available, replicated data store [72] makes this approach infeasible in many commodity deployments. In fact, Apache Flink's distributed dataflow runtime serves today as a feature-complete runner of Apache Beam[42], Google's open-source implementation of the Dataflow Model[55].

Microbatching: Stream micro-batching or batch-stream processing (e.g. Spark Streaming [12], Comet [83]) emulates continuous, consistent data processing through recurring deterministic batch processing operations. In essence, this approach schedules distinct epochs of a stream to be executed synchronously. Fault tolerance and reconfiguration is guaranteed out-of-the-box through reliable batch processing at the cost of high end-to-end latency (for re-scheduling) and restrictive model, limited to incremental, periodic immutable set operations. Trident [84], a higher level framework built on Apache Storm offered a form of processing guarantees through a similar transactional approach on predefined sets but executed on long-running data stream tasks. While fault tolerance is guaranteed with such techniques, we argue that high latency and such programming model restrictions make this approach non-transparent to the user and often fall short in expressibility for a significant set of use-cases.

4.7 Acknowledgements

The overall work on Flink's asynchronous epoch snapshotting protocol would not be as effective and impactful as it is today without the hard work and commitment of key people in the Flink community. First and foremost, I would like to thank Stephan Ewen, the contributions and influence of which address every small detail in Flink's production-proof state management. Furthermore, I would like to thank Gyula Fóra, Stefan Richter and Till Rohrmann among others for their involvement and implementation of key parts presented in this study such as Flink's backends. Finally, I want to thank Kostas Tzoumas and Flavio Junqueira for all the insightful discussions and the overall effort of the Flink community that keeps enriching Flink's state management with sophisticated optimisations, connectors and backends.

4.8 Summary: A Design Approach Perspective

We summarize the findings of asynchronous epoch-based commits and their applications in Apache Flink in terms of the three of our core design principles.

[D1] Blocking-Coordination Avoidance There are two types of coordination identified in epoch-based stream execution with snapshots. First, we have the centrally coordinated phases of an epoch, i.e., issuying epoch change events and collecting acknowledgements. The snapshotting protocol is another internal coordination mechanism which allows different processes to aquire an epoch-complete global system state using a distributed algorithm. In both cases, blocking synchronization is fully avoided. In the case of master coordination, all communication steps are triggered asynchronously to the underlying execution. Furthermore, the snapshotting protocol does not impose any blocking synchronization since even during epoch alignment all tasks continue their regular execution concurrently.

[D2] Runtime Transparency When it comes to the snapshotting technique, it is fully transparent to the user program. Epoch markers and local snapshotting are handled transparently by Flink's runtime, making zero restrictions on its programming model. In comparison, Spark's microbatching technique initially restricted programmers to express continuous computation into RDD transformations on processing time windows [12] . However, today we observe that these practices are avoided and Flink state management principles are also adopted by the Spark community for continuous processing [85] within Structured Streaming [74].

[D3] Model Compositionality Epoch-based snapshots offer a level of compositionality, not achievable by any other stream processing approach in the past (including regular snapshotting methods). As we analyzed already in section this chapter snapshots can be used for reconfiguration, fault tolerance as well as building sophisticated application provenance schemes on actual stream processing executions. Other design approaches that relied on a fine-grained fail-recovery model [11, 9], which were analyzed before achieved fault tolerance but did not consider such a compositional, application-wide concept as the one of snapshots (e.g., used for snapshot isolation of external queries, complete migration or provenance of continuous applications).

CHAPTER

5

Window Computation Sharing

Windowing is both an integral and challenging high level construct in data stream computing. On one hand, stream windows cover the needs of scoping and bounding an otherwise unbounded computation, while on the other hand, they introduce big challenges such as expressing, executing and optimising evolving and “embarassingly-redundant” aggregations on top. In this chapter, we present an analysis of the primitives and execution strategies for stream windows, their inherent “sliding” nature and the implications of running aggregations on top. We then distill the minimum programming model requirements to execute generic windowing operations-as user-defined functions, and present a prototype design of a an optimised execution plan that can reduce the total cost by orders of magnitude than the state of the art, integrated in Apache Flink.

5.1 Introduction

Windowing is a first class citizen of most stream processing systems [2, 86, 12, 87, 39]. Those systems typically support a set of restricted, predefined primitives in their programming model to construct time- and count-based windows of various forms (e.g., sliding, tumbling, hopping). This set of primitives can serve a variety of use cases that involve recurring aggregation on fixed periodic intervals and can be trivially optimised by the stream processor.

In contrast, many general dataflow programming systems do not provide predefined windowing primitives [9, 20], but encourage users to compose windowing via user-defined operators, often based on complex business logic. The main challenge of these use cases, is that they hide their semantics from the system and hinder optimization opportunities for efficient execution. This forces aggregation of non-periodic windows to rely on semantics-agnostic, best-effort aggregation data structures such as binary pre-aggregation trees [32, 15], that operate unecessarily at high memory and computational costs.

Seminal works in the past aimed at reducing redundancy in sliding window aggregation through sacrifing any means of compositionality and enforcing restricted, statically defined windows [14, 13, 88]. These restrictions are a major drawback for adressing the modern needs of composing general user-defined windows, needed to express punctuations [89], sessions [21], snapshots [90] and other types of data-centric computation on streams. We therefore, identify a case where assumptions at the execution level pose restrictions to a user-level programming model and thus disallow compositionality and programming model transparency.

The core objectives of this work are first, to identify the minimum programming model requirements in the context of sliding window aggregation. With this step we aim to lift assumptions from periodic windows to a broader class of sliding windows that can support efficient pre-aggregation with clear semantics. Second, we aim to design and implement an execution strategy that can incorporate the pre-aggregation of this broader class of windows as well as adequately supporting any user-defined window.To this end, we propose a programming model which enables expressing windows through UDFs and introduce the concept of User-Defined Windows (UDWs). We then exploit certain underlying properties of UDWs and devise a novel aggregate sharing technique that is applicable, not only to known periodic window classes, but to a broader class of windows with less space and computational costs compared to the state of the art. We further implement support for UDWs, their semantics, as well as our aggregate sharing technique on Apache Flink [43], showcasing a complete framework for streaming aggregations.

The outline of this chapter goes as follows: First, in section 5.2 we introduce and model the notion of sliding window aggregation queries in stream processing. Then, section 5.3 specifies the problem of aggregation sharing, generalizing existing state of the art techniques into broader categories and a common reference framework. Our solution to the problem is analyzed extensively in section 5.4 [1] followed implementation internals in section 5.5, an analytical complexity analysis in section 5.6 and experimental evaluation in section 5.7. Finally, the chapter concludes with aknowledgements (section 5.10), future work (section 5.9) and an extensive summary addressing our core design principles (section 5.11).

[1]Sections 5.4-5.7 include previously-published content [31] as-is.

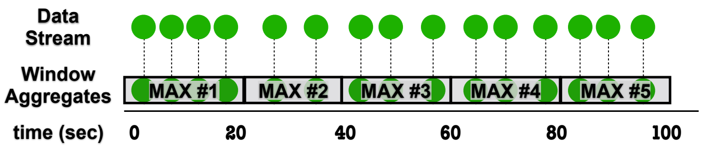


Figure 5.1: Tumbling windows of range 20sec.

5.2 Preliminaries

In this section we go through the concept of stream windows as well as partial aggregation semantics and their applications in order to motivate the need for a more compositional representation for sliding window aggregation.

5.2.1 Periodic Windows

The concept of sliding windows was introduced rather early in data stream management as a means of incorporating evolution in relational processing for unbounded data. The design of the Structured Query Language (SQL) had been initially driven by a clear goal, to be able to access and retrieve records in a DBMS, in a declarative and retrospective manner. SQL was originally retrospective since it was restricted to executing queries ad-hoc, i.e. compiling a complete answer on materialized tables and returning that answer back to the user. For example the SQL query below computes in an adhoc manner the maximum speed out of a table called "CarEvents" which yields a single answer (175kmh).

```
SELECT max(speed)
from CarEvents
```

```
> 175
```

Now, consider the case where we need to inspect the maximum speed throughout a live event such as a car race. A single aggregate in that case would not be particularly useful to analyze the evolution of the maximum speed during different highlights of the race. In that case, a window can simply extend the SQL query above with a duration or "range", upon each the query will return an answer as shown below (based on the CQL syntax [86]).

```
SELECT max(speed)
from CarEvents
[RANGE 20 Seconds]
```

```
> 172
> 171
...
```

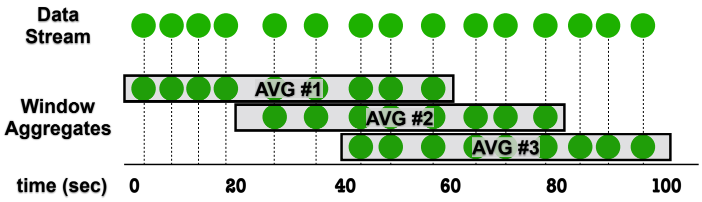


Figure 5.2: Sliding windows of range 1min and slide 20sec.

```
> 178
...
```

In this extension of SQL all queries are also known as standing queries, since the logic is being executed continuously. Windows with simply a "range" are also called "tumbling windows". In principle, for tumbling windows each record belongs to a single window, as depicted in Figure 5.1. Thus, the evaluation of each window can be performed trivially by grouping records and executing each aggregation (max(speed)) independently. However, general sliding windows add a challenging twist to the formula, namely the 'slide'. As an example, consider the following sliding window query (based in SQL-99 standard syntax):

```sql
SELECT AVERAGE(speed)
FROM CarEvents
[WATTR timestamp
 RANGE 20 second
 SLIDE 1 minute]
```

The "slide" represents "when" or "how often" a window has to be evaluated while including all records defined in its *range* in the computation of the maximum speed. In this sliding window example, there is an overlap of 40 seconds between each consecutive window as depicted in Figure 5.2. Therefore, a naive execution would result into redundant operations in its great majority.

5.2.2 Non-Periodic Windows

In addition to common periodic windows, stream processors such as Flink, Beam [42], IBM Streams [57] and Apex [39] provide built-in support for special sliding window types such as session windows [21] while also allowing users to implement the logic that discretizes streams via provided user-defined functions.

Session windows are a common example of windows with non-periodic characteristics. A session window starts with the first event received in a specified key (e.g., user id) and ends upon a specified duration of inactivity. For example, if no events of the same key arrive in ten minutes a session is closed. The Java code snippets bellow show how a session window with a gap of ten minutes is expressed

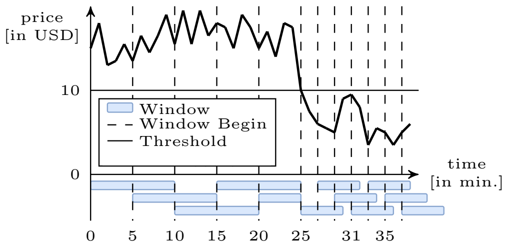


Figure 5.3: A dynamic window example: Reports become more frequent when the value of a stock is below 10$.

in Apache Beam and Flink's DSL respectively. On top, an aggregate function is evaluated per session window which computes the average speed.

```
//Beam
PCollection<CarEvent> input = ...
input.apply(Window<CarEvent>.into(
   Sessions.withGapDuration(Duration.standardMinutes(10))))
   .apply(new AverageSpeed());


//Flink
KeyedStream<CarEvent> input = ...
input.window(EventTimeSessionWindows.withGap(Time.minutes(10)))
   .apply(new AverageSpeed())
```

In the most general case, windowing can serve purely domain specific application logic, beyond sessions and thus, systems need to allow users to express how they want records to be assigned to windows and when to trigger their aggregation logic. In principle, windows often need to follow data trends, such as concept drift [91] for enabling continuous machine learning algorithms to operate on unbounded data. Another case that highlights the same need is when a continuously running system has to adapt the rate at which it notifies users for changes whenever human attention is needed and therefore, no static slide and range can incorporate that behavior.

Motivating Example: Consider a monitoring application for stock quotes that continuously receives records representing stock trades. Each record contains a volume (how many units where traded) and a price, per traded unit. A stock trader wants to see the volume-weighted average price of a stock over the last 10 minutes, reported every 5 minutes. However, when the stock's price falls below a certain threshold (e.g., the trader can buy the stock in a low price), the trader wants to receive an update every 2 minutes, with a weighted price average of the last 5 minutes. An example of such a monitoring dashboard is depicted in Figure 5.3. As specified by the trader, when the price falls below $10 on the 25th minute, the slide becomes more frequent (every 2min) and the window range becomes shorter (5min). More formally, the window definition goes as follows:

$$\text{window} = \begin{cases} \texttt{SLIDE=5min; RANGE=10min} \text{ if price} > \$10 \\ \texttt{SLIDE=2min; RANGE=5min} \text{ if price} \leq \$10 \end{cases}$$

This is a simple example of a purely data-driven, user-defined window (UDW) that dynamically changes its range and slide according to the incoming stream. The semantics of such windows have not been defined or supported by systems in the past. The major challenge for systems implementing such a sliding window computation is that no pre-defined execution strategy can be known apriori. The slide and range can change anytime which can make pre-aggregation challenging. We will revisit this example later in the section to describe our approach to User-Defined Windowing.

5.2.3 Data and Aggregation Model

Expressing sliding windows is only one side of the coin. From a system's perspective, sliding window aggregations are highly overlapping computations that can be approached differently based on the associated semantics. We will therefore make a clear data and computational model analysis to frame how sliding windows can be executed, under which assumptions and associated complexities.

5.2.3.1 Data Streams as Sequences

A data stream consists of records derived from a type T (e.g., `DataStream<T>` in Apache Flink). A convenient model to encapsulate a data stream, in this case, is the use of sequences. In fact, a stream $\bar{s}$ is a *sequence* $\bar{s} \in \mathtt{Seq}(T)$ where $\mathtt{Seq}(T)$ defines the set of all sequences that can be derived over T. Furthermore, we denote by $s_i = \bar{s}(i)$ the element at position i in data stream $\bar{s}$.

While a stream is unbounded, we are often interested in specific intervals to describe finite subsequences out of $\bar{s}$. An interval $R = [a, b]$ is a set of integers from a to b, $a \leq b$. $\bar{s}(R)$ is a *subsequence* of $\bar{s}$ where $\bar{s}(R) = \{s_i | i \in R\}$. In the following, we refer to $\bar{s}(R)$ as a *substream* and use $s[a, b]$ as a shorthand notation for $\bar{s}([a, b])$. We further denote the set of all substreams of type T as $\mathtt{Str}(T)$, where $\mathtt{Str}(T) \subset \mathtt{Seq}(T)$.

5.2.3.2 Discretization and Aggregation

We break down sliding window aggregation into two operations, Discretization and Aggregation. A `Discretize` operator transforms a stream $\bar{s} \in \mathtt{Seq}(T)$ into a sequence $\overline{w} \in \mathtt{Seq}(\mathtt{Str}(T))$ of (possibly overlapping) windows, where a window is a *substream* $w_i = s[l_i, r_i]$. In the following, we refer to an output sequence $\overline{w}$ as a

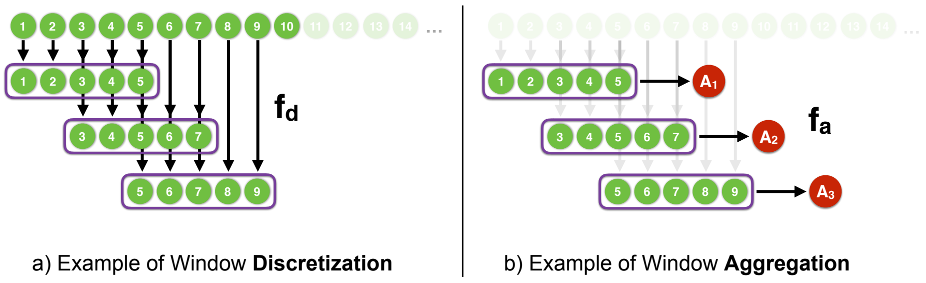

f_d
f_a
A1
A2
A3
a) Example of Window **Discretization**
b) Example of Window **Aggregation**

Figure 5.4: Discretizing and Aggregating a count window of range 5 and slide 2.

discretized stream and describe the discretization using a function f_{disc} :

$$\texttt{Discretize} : f_{disc} \times Seq(T) \rightarrow Seq(Str(T))$$

Windows can overlap, yet we assume that they maintain a FIFO order. More formally, for a pair of windows $w_i, w_j \in \overline{w}$, if $i \leq j$ then $l_i \leq l_j$ and $r_i \leq r_j$.

The `Aggregate` operator maps each window in the discretized stream to an aggregate value, given an aggregation function f_a.

$$\texttt{Aggregate} : (f_a : Str(T) \rightarrow T') \times Seq(Str(T)) \rightarrow Seq(T')$$

In the example of Figure 5.4 we show how the sliding window aggregation of a specific window (count range 5 and slide 2) is broken down into these two operations, alongside their application scope. The aggregation function f_a can support any aggregate measure from trivial distributive measures such as `SUM`, or algebraic such as `SUM`. In theory, holistic aggregate functions are applicable as well (e.g., median, mode and rank). However, in order to be able to support arbitrarily large windows most programming models disallow holistic aggregate functions to guarantee sublinear or constant bounds on the space needed to further describe sub-aggregates.

5.2.3.3 Partial Window Aggregation

Windows are finite but can be arbitrarily long, often containing millions of records. This makes the raw, individual aggregation operation after a complete discretization inefficient both in terms of space requirements and response speed. That is due to the fact that the aggregation will have to make another full traversal of all window elements after a full window has been triggered. Thus, it is generally preferable to incrementalize it.

More formally, an aggregation f_a can be decomposed into partial aggregate operations. We denote by A the type of partial aggregates. For example, in the case

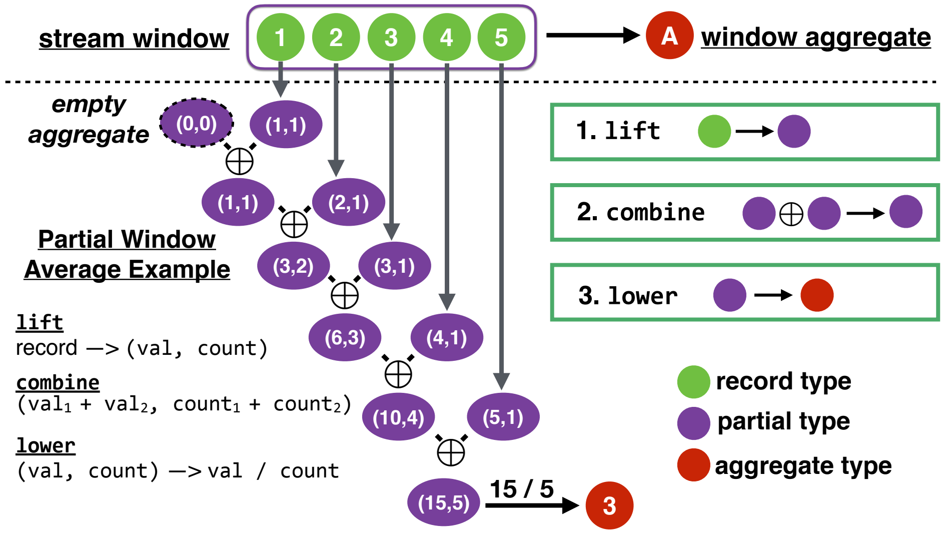


Figure 5.5: Partial Aggregation Example for Window Average.

of a SUM aggregation $A = \mathbb{R}$. A window aggregation function can be reformulated into three functions [15, 31], `lift` and `lower`, and a `combine` operator $\oplus$ [92]. as follows:

- `lift` : $T \to A$ maps an element of a window to a partial aggregate of type A.
- `combine` $\oplus : A \times A \to A$ combines two partial aggregates into a new partial aggregate (equivalent to a `reduce` function).
- `lower` : $A \to T'$ maps a partial aggregate into an element in the type T' of output values.

By using the above functions and operator, elements in the window are lifted to partial aggregates, further combined starting from the default (identity value) 1_A for type A, and finally mapped to an output aggregate value. For example, for window $s[1,3]$ the aggregation would be unrolled as:

$$f_a(s[1,3]) = \texttt{lower}(((1_A \oplus \texttt{lift}(s_1)) \oplus \texttt{lift}(s_2)) \oplus \texttt{lift}(s_3))$$

Generally, we denote the partial aggregate of a substream $s[i,j]$ as $P(s[i,j]) = 1_A \oplus \texttt{lift}(s_i) \oplus \ldots \oplus \texttt{lift}(s_j)$. The sole requirement for partial aggregation is to have an *associative* `combine` function so that aggregation can be used to evaluate a full window aggregate in discrete steps [32, 14, 15]. The final aggregate measure

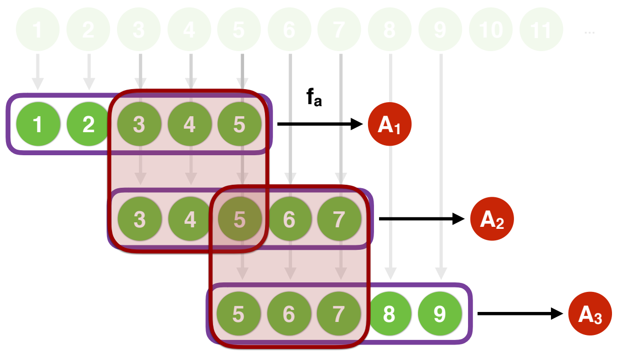


Figure 5.6: Visualization of redundant partial aggregates

can be distributive, i.e., composed out of multiple partially derived aggregate values such as average (using a sum and a count). An example of partial aggregation of a window is depicted in Figure 5.5. The goal in this example is to partially compute the average value out of a set of records with values 1 to 5. The invariant is that only one partial aggregate is kept in memory (initially an empty aggregate) that is being incrementally updated per record that arrives in a window. To compute an average two values are maintained in the aggregate type: a *sum* and a *count*. By using the `lift` function each record is first mapped into an aggregate type of its value and a count of 1. The `combine` function updates the partial aggregate with the new sum and count until all elements of a window have arrived. Then finally, the `lower` function transforms the aggregate into the window average, in this case this is 3.

5.3 Aggregate Sharing: Problem, Solutions, Limitations

Partial aggregation solves (as the name suggests) the general problem of sliding window aggregation only partially. That is, a naïve execution of an `Aggregate` operator across windows can lead to redundant computation. To demonstrate the problem assume we have a discretized stream $\overline{w}$. For any two windows $w, w' \in \overline{w}$ there is potentially an overlap $v = w \cap w' = s([l, r] \cap [l', r'])$. Computing all partials over $\overline{w}$ would yield:

$$\begin{aligned} P(\overline{w}) &= \ldots \cup P(w) \cup P(w') \cup \ldots \\ &= \ldots \cup (P(w \setminus v) \oplus P(v)) \cup (P(v) \oplus P(w' \setminus v)) \cup \ldots \end{aligned}$$

It is therefore clear from the formulation that if $v \neq \emptyset$, the redundant work done would be at least $P(v)$. The same type of redundancy also applies when executing the same aggregation on multiple sliding window queries on a shared data stream. For example, let $\overline{w}$ and $\overline{w}'$ be two discretized streams computed over a shared data stream $\overline{s}$. For windows $w \in \overline{w}$ and $w' \in \overline{w}'$ with overlap $w \cap w' = v$ and the same aggregation function, the amount of redundant work to compute $P(w \cup w')$ would also be at least $P(v)$. In Figure 5.6 we depict overlapping contents which in turn translate into redundant partial aggregates for the sliding window example introduced previously (Figure 5.4).

Existing optimisations of sliding window aggregation fall into two diverse categories: 1) Stream slicing for periodic windows and 2) General pre-aggregation for semantic-agnostic windows. Stream slicing is a family of memory-efficient aggregation sharing mechanisms which rely on the apriori knowledge (compilation-time) of where windows start and end. Whereas, general pre-aggregation techniques are agnostic of the exact formulation of windows, focusing on eager pre-aggregation sharing data structures that can potentially aid sharing across arbitrary windows at runtime. In the latter, we further analyze these approaches.

5.3.1 Stream Slicing

The concept of slicing is to take a data stream $\overline{s}$ and derive a small set of partial aggregates I, which can be used to compose full window aggregations. Hence, instead of keeping all records of active[2] windows $|W|$ in memory, we only keep their partial aggregates.

Stream slicing has been studied in the past mainly in the context of periodic windows [14, 13] which are the most trivial case of sliding windows known apriori. Slicing guarantees that for any active window $w = s[begin, end]$ there will be a sequence of partials over contiguous intervals from *begin* to *end*. For instance, consider windows $s[1,3]$ and $s[2,7]$ for which we want to apply slicing. We can derive $f_a(s[1,3])$ and $f_a(s[2,7])$ from a set I of three shared sliced partials as follows:

$$I = \langle \overbrace{P(s_1), \rlap{$\underbrace{}_{f_a(s[2,4])=\texttt{lower}((1_A \oplus P(s[2,3]))\oplus P(s[4,7]))}$}P(s[2,3])}^{f_a(s[1,3])=\texttt{lower}((1_A \oplus P(s_1))\oplus P(s[2,3]))}, P(s[4,7])\rangle$$

In practice, the two most popular slicing techniques that have been adopted by existing systems in the past are *panes* [13] and *pairs* [14] which are summarized further below.

5.3.1.1 Panes

The main idea behind *Panes* is that if we have a periodic window query with a fixed slide and a range it is trivial to break down the aggregation process into partials with a constant size, equal to the greatest common denominator of the respective *range* and *slide*. For example if we have a sliding window with a range of 9 minutes and a slide of 6 minutes the stream would be sliced and pre-aggregated into buckets,

[2]Windows are active as long as discretization is pending.

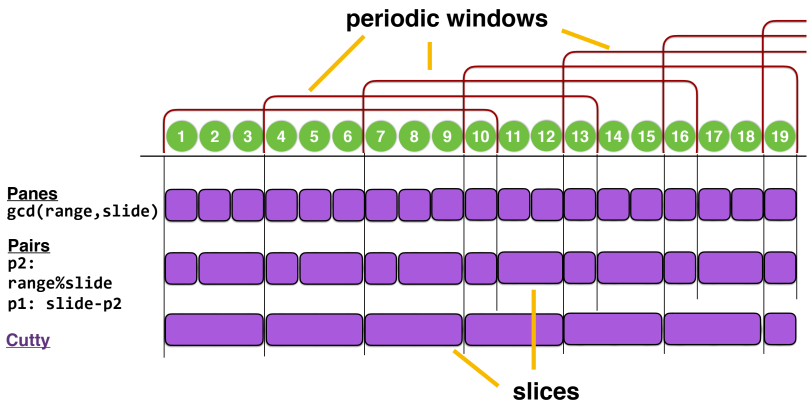


Figure 5.7: Example of different slicing techniques

each of which corresponds to 3 minutes of the ingested stream (greatest common denominator of 6 and 9).

5.3.1.2 Pairs

While being popular optimisation, *Panes* have been criticized [14] for their unbalanced performance that depends highly on the combination of range and slide. For example, a window of range 10 seconds and a slide of 3 seconds would break down to slices of a single second, no longer exploiting the amount of non-overlapping segments in a stream (as depicted in Figure 5.7). Instead, the *Pairs* technique [14] splits a stream into two alternating slices: $p_2 = \mathtt{range} \mod \mathtt{slide}$ and $p_1 = \mathtt{slide} - s_2$. This optimisation utilizes better non-overlapping segments in a stream compared to panes and is seamlessly applicable to any periodic window combination of range and slide. Intuitively, pairs "halt" the pre-aggregation of a partial only when a stream window starts or ends. This is visualized in Figure 5.7 where it results into a lower number of slices for the same window compared to using *Panes*. Contrary to *Panes*, *Pairs* has also been proposed for the case of multi-query aggregation sharing where a large sequence of shared slices is decided at compilation time across shared sliding window aggregation queries in the same operator [14].

5.3.1.3 Technique Analysis

Slicing undoubtedly offers the best known space and partial-aggregation complexity in sliding window aggregation as follows:
Space: Pairs, the most efficient slicing technique known so far, produces two partials per active window in a stream. Thus, at any given time, for $|W|$ active sliding windows we have a space complexity of $O(2|W|)$.

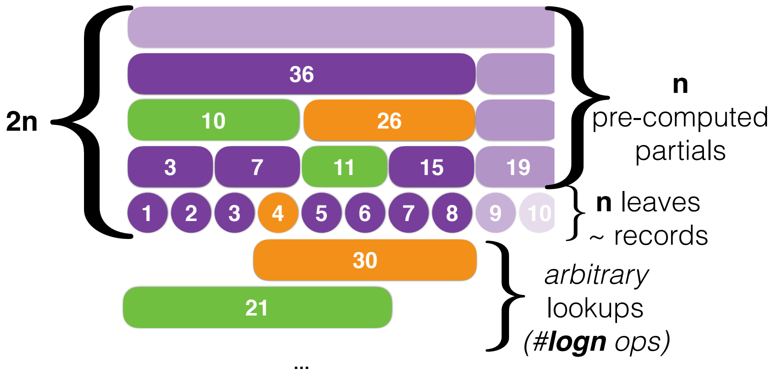


Figure 5.8: Example of a general pre-aggregation tree

Update (partial aggregation): The update complexity corresponds to the number of combine calls invoked per record for partial-aggregation. For all slicing techniques this is a constant of $O(1)$ since only one such operation is necessary per record.
Lookup (final aggregation): The "lookup" or final aggregation cost corresponds to the number of combine calls that are needed to compute a full window. This is equivalent to the number of partials generated which can be no more than $2|W|$ at any given time in the case of Pairs. Hence, final aggregation has a $O(2|W|)$ best known computational complexity.

Despite its performance, the applicability of slicing has been limited to periodic window queries since this is the only type of windows for which their beginning and end are known apriori. Later in this chapter we introduce the *Cutty* slicing technique [31] which avoids this strong assumption by letting user-defined windowing functions signify to the system during runtime when windows start. We will further show how this technique yields a minimal number of partials and improving further over *Pairs* in terms of required number of partials stored (as hinted in Figure 5.7).

5.3.2 General Pre-aggregation

The main incentive of general pre-aggregation techniques is to be able to allow for aggregate lookups on arbitrary substreams as depicted in Figure 5.8. When windows are non-periodic, with possibly unknown semantics at compilation time, general pre-aggregation serves as the only known "just in time" optimisation. It is, in essence, an "eager" pre-aggregation approach to generating and storing partials that can be potentially used later for evaluating the aggregation value of full windows when that is required. Earlier work on general pre-aggregation such as *B-int* [32] defines an appropriate binary-tree data structure that maintains partials and higher order partials which can be eagerly pre-computed during stream ingestion.

The application of B-int was meant to be fast aggregate retrieval for ad-hoc stream queries (i.e., using a declarative query language such as CQL [86]), however, the applicability of general pre-aggregation makes such techniques convenient for aggregating continuous sliding windows without a known range or slide or other periodicity assumptions. The Reactive Aggregator by IBM Research [15] exploits the properties of *B-Int* and introduces FlatFat: a fixed size circular heap binary tree of higher order partials that "slides" together with the records of the stream.

Example: The usage of FlatFat is demonstrated in the example of Figure 5.8 where a simple `sum` per record index is pre-computed. The pre-aggregation tree (which can be maintained in a heap) holds up-to-date aggregates from raw records ingested so far. Each partial-aggregate per input record is stored into one of the N leaves of the data structure , followed by $\log N$ combine calls to all parent partials in the tree in order to update all higher order partials that contain that part of the stream. In the same example, the consumption of record 4 would update and store the sum partials on the following set of substreams: $s\{[4], [3, 4], [1, 4], [1, 8], [1, 16]\}$, given a capacity of $N = 16$ record aggregates. At any time, the application can request an aggregate on an arbitrary substream such as $s[4, 8]$. In that case, the result is the combination of two pre-computed partials: $sum(s[4])$ and $sum(s[5, 8])$ as shown in the figure.

This approach 1) guarantees a $\log N$ upper bound on the number of operations for a full window aggregation and 2) bounded overall space of $2N$ where N stands for the size of the longest substream needed to compute currently active windows. Throughout this section we will also refer to this technique as "eager pre-aggregation", in contrast to all other techniques (including plain slicing) which employ a lazy approach to compute the full window aggregation from fine grained low partials (e.g., records or predefined slices).

5.3.2.1 Technique Analysis

Eager pre-aggregation offers fast retrievals of arbitrary windows on a stream at the cost of additional space and incremental update computation requirements. That is due to the fact that every time a new aggregate is added to the binary tree a sum of $\log(N)$ additional partial aggregations need to be employed in order to update all higher order aggregates of the tree (given N: the number of active records/leaves in the tree). In summary, the runtime costs of employing eager-aggregation, which are also visualized in Figure 5.8 are the following:

Space: The space requirements in case of FlatFat is exactly $2N$, the size of its allocated heap space needed for all stored partials.

Update/Lookup: Both update and full window lookup have $O(\log N)$ computa-

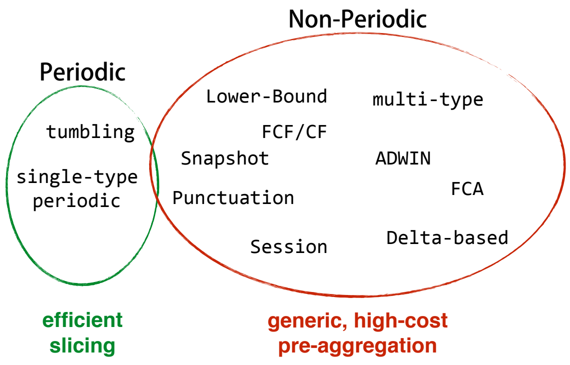


Figure 5.9: Applicability of slicing and general pre-aggregation

tional complexity. In the case of updates the complexity is fixed ($\log N$) against slicing which typically involves a single aggregation per record.

All these costs pose an interesting trade-off when this technique is employed per-record in a data stream, which often results in more operations than a naive execution of each redundant window aggregation separately since almost always. As a result, it is likely that if general pre-aggregation is de-facto applied, its runtime cost would never be amortized across a full run of a continuous application. In fact, excluding the case of windows that slide per record it is true that $N >> |W|$ and thus, slicing is ultimately preferred if it can be applied. Nevertheless, the power of general pre-aggregation lies at the observation that aggregation sharing can be employed in a window-agnostic manner, thus, covering a large space of non-periodic window types used within research and industrial applications.

5.3.3 Limitations and Motivation

The dichotomy between efficient aggregation on periodic sliding windows (i.e. slicing) and memory-demanding, best-effort aggregation as the fallback solution has been a burden for data stream programming models. On one hand we have the majority of models restricting expressibility to periodic windows to enable efficient aggregation (e.g., Spark D-Streams [12]). While on the other hand, we can only enable universal expressibility at the high cost of maintaining pre-aggregation data structures on raw streams. In Figure 5.9 we show examples of widely used windows categorized by the applicability of aggregation sharing techniques. Evidently, there is a need for an efficient aggregation scheme that can allow users to freely express more dynamic or complex application-driven windows while being able to distill the right primitives to enable efficient aggregation sharing. Such an approach would not trade off performance for model compositionality and it would further allow for systems to employ pre-aggregation transparently.

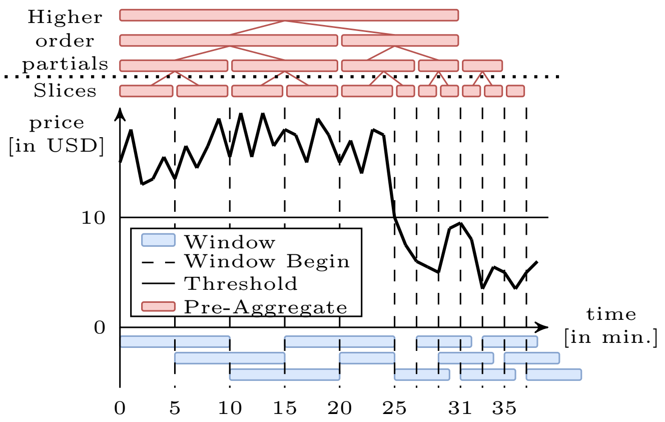


Figure 5.10: Motivating example including slices and higher-order partials: Reports become more frequent when the value of a stock is below 10$.

5.4 Cutty: A Hybrid Approach

We view slicing and general pre-aggregation as orthogonal techniques that can be potentially combined to support a wide variety of stream windows for efficient aggregate sharing. At the same time we seek for the minimum, transparent programming model requirements to enable slicing for a wider variety of sliding windows than periodic. These are the driving needs behind our design. In this section, we present the intuition and formal specification of deterministic user-defined windows as well as *Cutty*, an aggregation framework that can optimize slicing when that is achievable.

5.4.1 Approach Intuition

In 5.2.2 we presented different types of non-periodic windows followed by an example of a use-case where users need to specify sliding windows that change according to the value of a stock. Let us revisit this example with the notion of slicing and general pre-aggregation in mind. As depicted in Figure 5.10, the exact specification of sliding windows (e.g., range and slide) might be known only during runtime, however, a system does not necessarily need to have full knowledge of these semantics beforehand as long as it knows when is the right time to complete the pre-aggregation of each slice. Our example case hints that this is possible as long as the system knows at ingestion-time when a record marks the beginning of a window. At that point it can store the result of its partial aggregation so far (active partial) and start anew with a new pre-aggregation. This approach guarantees that we store just enough slices to compose any window that is known at runtime. When

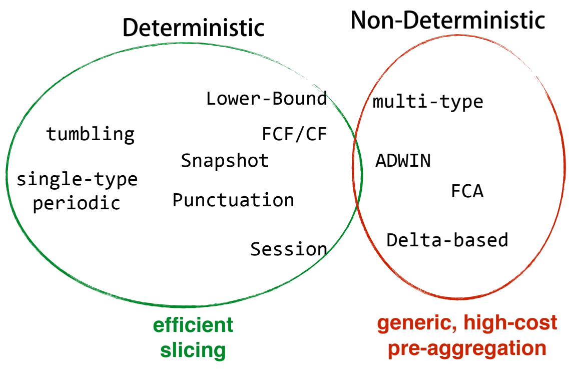


Figure 5.11: Extending applicability of slicing with Deterministic UDWs

we reach a record that marks the end of a window, we can combine the current active partial, together with previously stored partials (slices) to get the aggregate of the full window. Therefore, at any given point we need to maintain no more slices than there are active windows $|W|$.

General pre-aggregation data structures such as FlatFat are a convenient means of speeding up the final window aggregation by sharing higher-order partials that go beyond the scope of slices. Hence, we foresee a benefit of exploiting general pre-aggregation data structures on top of slices in order to reduce final window aggregation complexity from $O(|W|)$ to $O(\log|W|)$. The memory cost of maintaining these data structures on raw streams can get very high, however, on top of slices the space complexity becomes a constant of $2|W|$. In Figure 5.10 we visualize how slices and their corresponding higher-order partials can both serve the computation of arbitrary windows in the previous example . Given that the technique considers individual windows and not queries, it can also be utilized in a multi-query aggregation sharing environment where multiple sliding windows share the same aggregation on an underlying stream.

5.4.2 User-Defined Windows

We distinguish two general classes of User-Defined Windows (UDWs), namely *deterministic* and *non-deterministic*, which are explained below.

Deterministic UDWs: We coin as deterministic all windows for which we can apply slicing efficiently as described previously, while non-deterministic are all the rest for which we cannot. Deterministic windows can be declared with a user-defined *discretization function* f_{disc}.

The concept of deterministic windows stems from the observation that all it takes to achieve optimal slicing is not the *apriori* knowledge of the periodicity

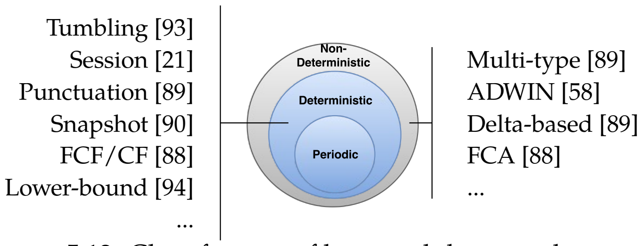


Figure 5.12: Classification of known sliding window types.

of windows (if any) but the *runtime* knowledge of where a new window starts. Intuitively a window function is deterministic if at the time that it processes a record, it can decide whether that records marks i) the beginning of a window or ii) the end of a window. For instance, a periodic count window of fixed range and slide, is deterministic, since when a record arrives, an internal counter can affirm whether a new window begins with that record. Similarly, a punctuation window is deterministic, since (by definition) a punctuation marks the beginning of a window. Formally, a deterministic window function f_{disc} is defined as follows:

$$f_{disc} : T \rightarrow \langle W_{begin} : \mathbb{N}, W_{end} : \mathbb{N} \rangle$$

where for each record $r \in T$: *i)* W_{begin} is the number of windows beginning with r and *ii)* W_{end} the number of windows ending with r.

As an example, consider the following deterministic function for declaring periodic count windows of fixed range and slide:

$$f_{disc}(s_k) = \begin{cases} W_{begin} : 1 \text{ if } k \texttt{ mod slide} = 0\text{, else } 0 \\ W_{end} : 1 \text{ if}((k - \texttt{range}) \texttt{ mod slide}) = 0\text{, else } 0 \end{cases}$$

In fact, periodic windows which are supported by all existing event processing systems and query languages (e.g. CQL [86]), are subsumed by deterministic.

Figure 5.11 depicts the extended set of known window types that can support slicing in the context of deterministic windows. Furthermore, Figure 5.12 provides an overview of the window classes along with a categorization of known window types found in the literature. Tumbling windows are periodic and thus, deterministic. Session and snapshot windows by definition begin with the first record of the session that marks the window's beginning and end with a timeout record which is injected in the system (e.g., a watermark). Moreover, lower-bound landmark windows begin from a given landmark record, that marks the beginning of a record and end upon a punctuation or a predefined length.

Non-Deterministic UDWs: Intuitively, non-deterministic windows cannot declare immediately whether a record begins a window or not, i.e., they need to examine more records in order to take such a decision. As a result, slicing for non-deterministic windows is not possible and we have to fall back to general

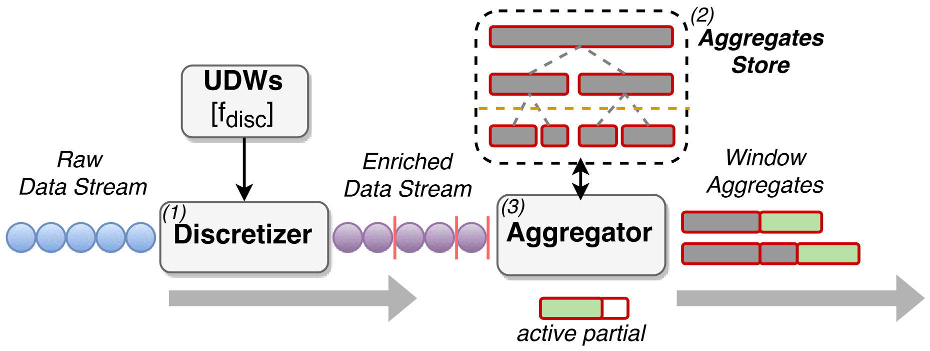


Figure 5.13: Architectural overview of Cutty.

pre-aggregation techniques [32, 15]. A typical example of a window which is non-deterministic is a window which every 5 seconds outputs the last 10 records of the stream. If we assume that a record r_1, arrives during the first second of a window and the next second, another 10 records arrive, r_1 will not be part of the next window. Thus, if the rate of the stream cannot be known in advance, such a window cannot be deterministic; the window function cannot specify whether a record is going to be part of the next window at the very moment of the record's arrival.

5.4.3 Overview of Cutty

Cutty is a framework for slicing and pre-aggregating multiple overlapping windows, implemented on Apache Flink. By using the *Cutty* framework instead of a naive execution of sliding window aggregation, we can achieve high sharing and minimal redundancy across any sliding window expressible with a deterministic UDW. The framework consists of three main components, as depicted in Figure 5.13:

- A *discretizer* operator that enriches the incoming data stream with windowing information dictated by a one or a set of UDWs that have a common aggregation measure (e.g., SUM, AVG). The enriched stream drives the pre-aggregation decisions and aids the aggregator with the bookkeeping of active windows.

- A *shared aggregation* operator which slices the enriched input stream and evaluates full window aggregates, backed by an aggregation store at its state.

- The *aggregation store*, a special data structure based on FlatFat [15] that maintains pre-computed higher order partials allowing efficient aggregate lookups for arbitrary window intervals.

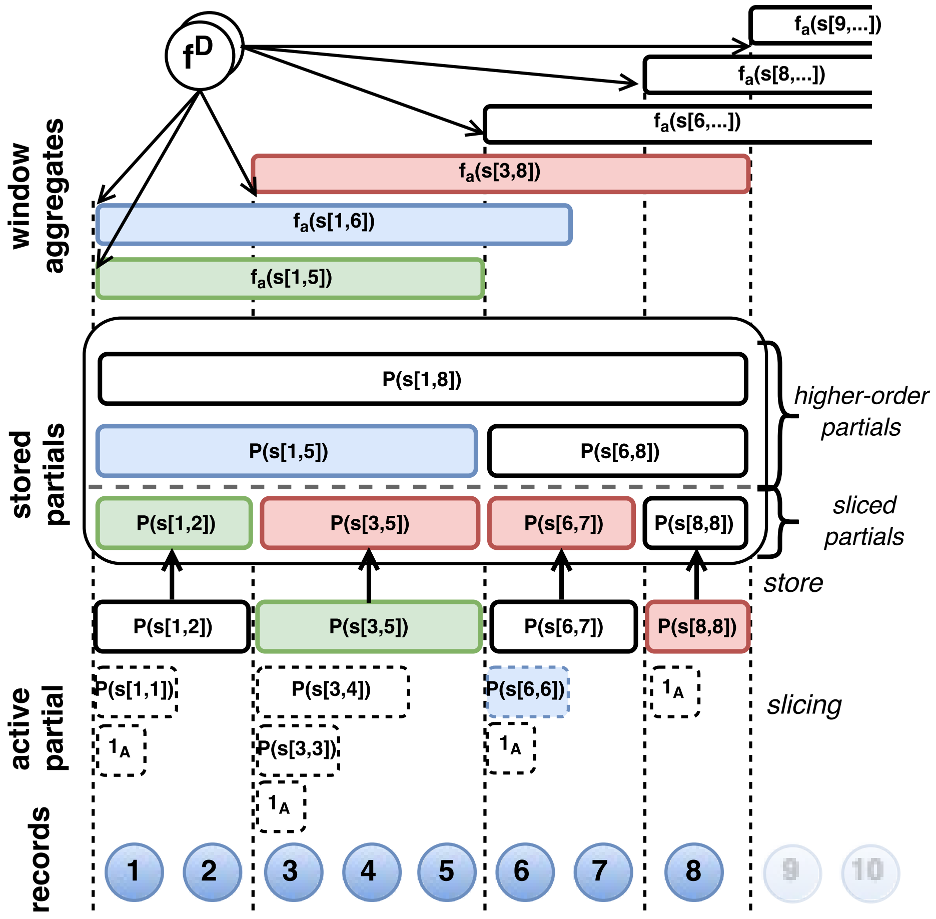


Figure 5.14: An example of Cutty on deterministic windows.

In the rest of the section we will demonstrate the execution of *Cutty* followed by section 5.5 where we analyze each of the design choices made in the framework.

A Thorough Example. Figure 5.14 depicts a full example of the execution of Cutty for aggregating a set of deterministic UDWs. As the set of the deterministic UDWs dictate (f_{disc}'s at the top), partial aggregation is applied incrementally only within the intervals that mark the beginning of windows (dashed vertical lines). At the beginning of every interval, the active partial resets to the initial value 1_A and maintains the current pre-aggregate until it forms a complete atomic slice (in this example, $(P(s[1,2]), P(s[3,5]), P(s[6,7]), P(s[8,8]))$. Before the active partial resets, its value is stored for further reuse (central rectangle in the figure). When a window ends we have everything to compute the full aggregation from the partials. For instance, in the case of $f_A(s[3,8])$, upon getting notified at the consumption of s_9 that window $s[3,8]$ is complete we are ready to compute the full aggregation. We can derive the full window aggregation by re-using all precomputed slices $P(s[3,5]), P(s[6,7])$ and $P(s[8,8])$ which are already stored.

Sharing Higher-Order Partials. From the example in Figure 5.14, we observed that, simply sharing sliced partial aggregates does not eliminate redundancy when computing full window aggregations. For example, with a typical lazy evaluation slices $P(s[1,2])$ and $P(s[3,5])$ would have to be combined twice: once for computing $f_a(s[1,5])$ itself, and once for computing $f_a(s[1,6])$. Instead, if $f_a(s[1,5])$ was stored, $f_a(s[1,6])$ could be computed by $f_a(s[1,5])$ together with the current partial. In principle, the higher the number of overlapping windows, the more reduce calls are repeated for each complete window lookup operation. To deal with this issue we can exploit an eager pre-aggregation strategy to incrementally pre-compute higher-level aggregates. However, in contrast to the original usage of eager pre-aggregation [15, 32] on raw record streams, we apply this technique on slices to produce higher order reusable partials. This way, we gain a logarithmic decrease of each complete window aggregation at a low additional memory footprint. Furthermore, the same strategy can be re-used as a best-effort fallback solution for non-deterministic UDWs, where slicing cannot be applied.

5.5 Internals and Implementation of *Cutty*

In this section, we dive deeper into the internal design of the Cutty framework by explaining further each of the core components: the discretizer, shared aggregator and its aggregate store.

5.5.1 A Shared Discretizer for UDWs

Cutty's discretizer is able to multiplex multiple UDWs, coming from one or multiple applications. UDWs can be added at job compilation time all once or incrementally, by reconfiguring the job to load new UDWs. Its basic functionality is to consult all the UDWs per input record, enrich that record with extra information, as explained below, and pass it downstream to the aggregator.

Deterministic UDWs. As we have seen in subsection 5.4.2, deterministic functions specify whether a record begins or ends a window, exactly at the moment of the record's arrival. This gives the discretizer the ability to give hints to the aggregator, alleviating the aggregation process from the need for window management and

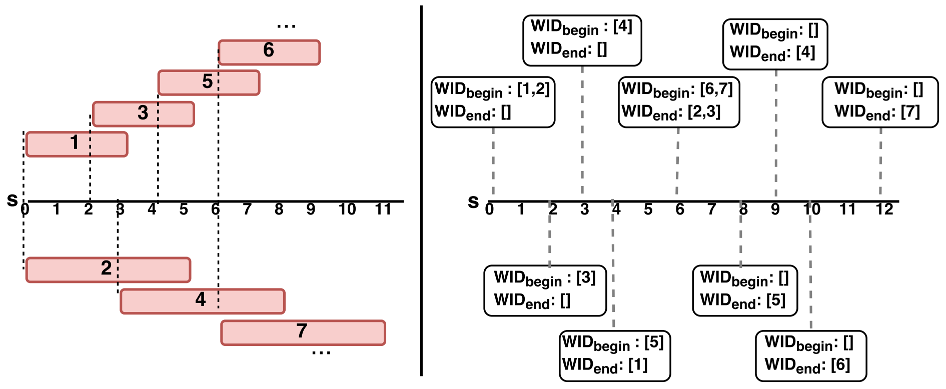


Figure 5.15: Mapping multiple UDWs to discretization events.

bookkeeping. Hence, the aggregator simply operates on a marked stream, without any knowledge about the specifics of the UDWs.

The discretizer associates a set of window begin/end identifiers with each incoming record. These identifiers are used by the aggregator to apply slicing incrementally. Consider for example the shared discretization of windows produced by two UDWs as shown in the example in Figure 5.15. The upper UDW has a range of count 4 and slide 2 while the one in the bottom has a range of count 5 and slide 3. The dashed vertical lines mark the begin of each individual window, as indicated by the UDWs. The discretizer consults the UDWs and injects "window begin" markers in the stream (as depicted on the right). For instance, on record 0, there are two window begins, namely windows 1 and 2. Similarly, the discretizer consults the UDWs and enriches the stream with "window end" markers. In our example, window 1 ends with record 3 (and indicated by the UDW upon processing record 4). Formally, the discretizer injects window identifiers as follows:

$$r : T \mapsto \langle r : T, \mathrm{WID}_{begin} \subset \mathbb{N}, \mathrm{WID}_{end} \subset \mathbb{N} \rangle$$

where WID_{begin} and WID_{end} are derived from the UDWs and represent each unique window that, respectively, begins (inclusively) or ends (exclusively) with record r.

Non-Deterministic UDWs. Non-deterministic windows cannot indicate whether the current record of the stream begins a new window. To this end, non-deterministic UDWs have to indicate possibly expired records that should be removed from the head of the current window, and notify when the window has to be emitted. Since the shared discretizer operates on multiple UDWs at the same time, it has to i) derive the records that expire across *all* UDWs i.e., the intersection of records that all UDWs have declared as expired ii) store and track the begin of each active window.

Non-deterministic windows in our aggregator architecture are handled by the underlying aggregate store [15].

5.5.2 Shared Aggregation

The shared aggregator is an operator that consumes the enriched streams of the discretizer and generates window aggregates. Consecutive input records are aggregated, at a low cost, into an active partial aggregate maintained at its managed state, whereas, window "begin" and "end" indicators in the enriched stream are used to add and retrieve partials (slices) stored at its state to evaluate full windows.

Slice State Interface: There are three operations supported at the aggregator's state to manage pre-computed slices for adding, looking up and purging slices as follows:

- `add(partial_id, partial)`: Adds a partial at the end of the store where `partial_id` is an identifier for the provided partial.
- `lookup(from, to)`: Computes result of $\mathsf{P}(s[\mathsf{from}, \mathsf{to}])$. Most of the time the aggregator is interested in looking up a full aggregation starting by `from`. In that case, we will use the shorthand call `lookup(from)`.
- `purge(partial_id)`: Removes all given partials from the store up to `partial_id`.

As explained further in 5.5.4, we considered two evaluation strategies for the store, a *lazy* using a circular fixed-sized array, and an *eager* which builds on FlatFAT [15] using a pre-allocated circular buffer backed by the local JVM heap. For the rest of this section, it should be assumed that the store follows an eager strategy unless stated otherwise.

Aggregation Logic: The functionality of the Cutty aggregator is summarized in algorithm 8. The aggregator maintains a single active partial on which it applies incremental aggregation per record arrival. Effectively this is an execution of stream slicing, where each slice spreads between records that mark consecutive window begins. In case a new window begins (which occurs when the set $\text{WID}_{\mathtt{begin}}$ is not empty) the aggregator has to start a new partial pre-aggregation. In that case, the current active partial is stored in the aggregates store and the active partial resets back to its initial value (1_A). In any other case the aggregator simply applies a single `combine` operation to update its active partial. Mind that the aggregator keeps track of the first record of every active window since it has to be aware of the specific range to aggregate when a window ends. For every window that ends in $W_{\mathtt{end}}$, the aggregator combines its active partial with the stored partial of the interval that starts at the beginning of the window.

Algorithm 8: Cutty Shared Aggregation

```
1: Upon $r_i, WID_{begin}, WID_{end}$
2:     if $WID_{begin} \neq \emptyset$ then
3:         store.add($i$, partial);
4:         partial $\leftarrow 1_A$;                              ▷ reset active partial
5:         foreach $w \in WID_{begin}$ do
6:             begins[$w$] $\leftarrow i$;                        ▷ mark begin for $w$
7:     foreach $w \in WID_{end}$ do
8:         start $\leftarrow$ begins[$w$];                        ▷ retrieve begin of w
9:         begins.remove($w$) ;
10:        store.purge(min(begins));                              ▷ gc
11:        emit $\langle w \mid$ lower(store.lookup(start) $\oplus$partial) $\rangle$;
12:    partial $\leftarrow$ partial $\oplus$lift($r_i$)          ▷ online aggregation
```

Storage Costs. A very important feature of Cutty aggregation for deterministic UDWs is that it generates a *minimal* amount of sliced partials needed for any shared window computation. In the worst case, Cutty stores only as many partials as the number of active windows, compared to other slicing techniques [14] that generate twice as many partials. The main idea lies at the observation that only windows that end at a specified record r_i would need to aggregate up to i. Since no other windows would ever need to start or end at this index later, incremental aggregation can continue until it reaches a record which starts a new window.

For non-deterministic window aggregations, it is not possible to apply slicing due to the limited knowledge of the active windows. However, in that case Cutty utilizes the store, thus, bounding its performance to the current state-of-the-art approach [15]. Expired record removals and full window aggregates are executed based on the discretizer-injected information. Partial aggregation sharing is therefore achieved only via the eager strategy of the aggregate store.

5.5.3 Sharing for Heterogeneous Sliding Windows

The slicing logic of Cutty applies for both single and multiple multiplexed windows. To the best of our knowledge, Cutty is the first general slicing technique that combines deterministic windows defined on different metrics to share sliced pre-aggregate. This is because UDWs require lower level primitives (i.e. window begin and end) than application semantics (i.e., sessions, time ranges and slides etc.). Using Cutty, all window begins are mapped to a concrete record in the stream at runtime, which is the first record of the window. It doesn't matter which measures are used in the queries to define windows; knowing at which records windows

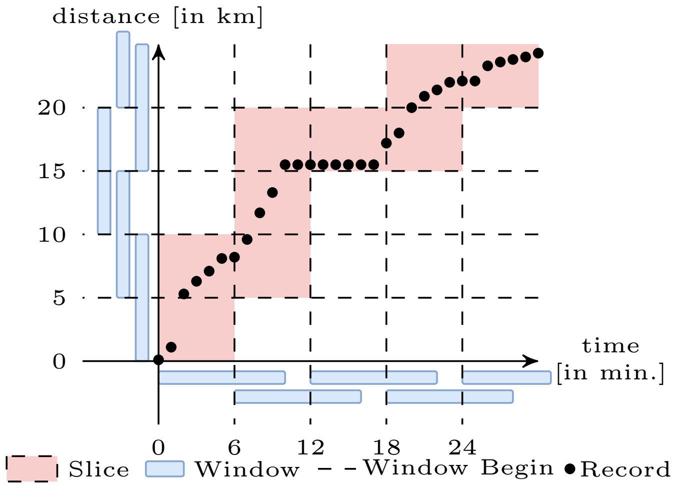


Figure 5.16: Incremental slicing for multiple queries which define windows over different measures.

begin, allows us to share pre-aggregates among them. Figure 5.16 illustrates how slices are created when two queries use different measures, namely time and distance. Imagine a vehicle which produces a stream of reports consisting of the current timestamp and mileage. We now define two periodic sliding window queries, which can share pre-aggregates: *i)* $\mathtt{Slide} = 6$sec.; $\mathtt{Range} = 10$sec. (depicted on x-axis) *ii)* $\mathtt{Slide} = 5$km; $\mathtt{Range} = 10$km (depicted on y-axis). Note that, for the sake of simplicity, the queries are periodic. However, Cutty can apply slicing and sharing on *any* combination of deterministic windows.

5.5.4 Aggregate Store Internals

The aggregate store implements the three operations needed by the aggregation logic, namely *add, lookup and perge*. For its default, eager evaluation strategy we based our implementation on FlatFAT [15], which we have extended to support sharing partial aggregates among multiple queries. As in FlatFAT, partials in our store are stored in a pre-allocated heap-based data structure that is pointer-less. The data structure resembles a "*sliding binary tree*" that pre-computes high level aggregates incrementally. In its circular heap space, single-hop tree traversals (e.g. *parent*, *rightChild*) can be executed in $O(1)$ time without the need for look-ups. In this part, we summarize all additions and considerations in the store internals while omitting several non-critical details that can be further studied in the Reactive Aggregator (RA) [15], the original aggregation framework of FlatFat.

Multi-window Processing. *Cutty* takes care of generating shared partials, produced by slicing multiple deterministic windows accordingly, and then stores them by invoking the `add(partial_id, partial)` operation. Stored partials (as opposed to the records themselves in [15]) serve as the leafs of the tree and are uniquely addressed by their original id. To allow this we enriched the data structure with additional mappings $h(partialID) \rightarrow i$ where $i \in \mathbb{N}$ is the heap index for that partial. When required, the aggregator has to look-up for aggregates on intervals within the partial id space and retrieve them via a `merge(from, to)` operation. Both `merge` and `add` operations employ a bottom-up heap traversal for pre-computing (`lookup`) and evaluating an aggregate range (`add`). The traversal yields an upper bound complexity of $O(log_2(n))$ in both cases, where n is the number of leafs. For *Non-deterministic* windows we implement an identical strategy to RA [15], by appending every individual record to the data structure and applying look-ups in the granularity of records.

Memory Management. Given that the number of active windows and therefore stored slices can only be known in runtime we can only speculate the size of the allocated heap space and resize the data structure when that is necessary via the use of heuristics. When the heap space is fully utilized upon an `add` operation, we double the array's capacity. This is a common strategy, also considered in the original implementation of FlatFAT [15]. Similarly, when a `purge` operation leaves behind an under-utilized heap down to a third of its full capacity, we half its capacity. Each storage resize invokes exactly $2n + 1$ heap operations. That means that if we need to store x partials we would pre-allocate a capacity $n = 2^k$ for $k = min\{m \in \mathbb{R} | m \geq log_2(x)\}$.

Lazy Implementation. The aggregates store also implements a *lazy* aggregation mode. This means that, when operating in this mode, no higher-order partials are precomputed. In this case, only first order partials are stored in the circular buffer (corresponding to leafs in the eager strategy). Thus, `add/purge` and `lookup` operations have $O(1)$ and $O(n)$ complexities respectively for n partials. For the sake of simplicity and comparability we preserved the same eager memory management strategies that were explained above.

5.6 Analytical Comparison

So far we have introduced different window classes and aggregation techniques for each class. In order to put everything into perspective we will examine worst case and amortized spatial and computational complexities exhibited by Cutty compared to other known techniques in aggregate sharing.

Analysis Scope. We deliberately focus on a single periodic window aggregation, with a fixed range r and slide s, since this is the single common denominator of all supported window classes. Our evaluation in section 5.7 includes experimental

	Space		Pre-Aggregation			Lookup	
	Lazy	Eager	Lazy	Eager	Eager (Amortized)	Lazy	Eager
Cutty	$\lceil r/s \rceil + 1$	$\lceil 2r/s \rceil + 1$	1	$O(\log(\lceil r/s \rceil))$	$(\log(\lceil r/s \rceil)-1)/s + 1$	$\lceil r/s \rceil + 1$	$\log(\lceil a \rceil)$
Pairs[14]	$\lceil 2r/s \rceil$	$\lceil 4r/s \rceil$	1	$O(\log(\lceil 2r/s \rceil))$	$2(\log(\lceil 2r/s \rceil)-1)/s + 1$	$\lceil 2r/s \rceil$	$\log(\lceil 2r/s \rceil)$
Panes[13]	$\lceil r/gcd(r,s) \rceil$	$\lceil 2r/gcd(r,s) \rceil$	1	$O(\log(\lceil r/gdc(r,s) \rceil))$	$\log(\lceil r/gdc(r,s) \rceil) + gdc(r, s) - 1$	$\lceil r/gcd(r,s) \rceil$	$\log(\lceil r/gcd(r,s) \rceil)$
RA [15]	r	$2r$	1	$O(\log(r))$	$\log(r)$	r	$\log(r)$

Table 5.1: Complexities of Cutty and the state of the art over aggregating a periodic sliding window.

analysis of multi-window scenarios. We cover all periodic window-centric pre-aggregation techniques, namely *panes* [13] and *pairs* [14] and the state-of-the-art general window aggregation techniques, *B-Int* [32] and *RA* [15].

5.6.1 Complexity Analysis

Table 5.1 summarizes all complexities, covering worst case memory demands in terms of number of stored partials, as well as pre-aggregation (computation per record) and lookup (computation per full window) in respect of reduce calls. We further decouple slicing methods from aggregate store strategies, i.e. eager and lazy. Evidently, general aggregation strategies conceptually slice a data stream by producing a partial per record. From the table it is clear that in all cases, the eager strategy achieves better merge performance while increasing memory and computational demands for updates. Furthermore, Cutty exhibits the most efficient execution both in terms of space and computation. This is because slices in Cutty solely correspond to the number of active windows at any given time. *Pairs*, on the other hand, require double the memory and computational resources as a side-effect of slicing twice more partials. *Panes* is the least efficient technique for periodic windows due to enforcing finer slicing granularity which corresponds to the smallest possible slice during a full window pre-aggregation. Finally, *RA*, as expected, has significantly more memory and computational demands than periodic window pre-aggregation techniques since it is agnostic of window semantics and thus, operates at the granularity of each individual record.

5.6.2 Amortized Costs

An amortized study of the operation costs of all these techniques with an eager strategy unveils further benefits for Cutty. Intuitively, the worst case update complexity in a full window aggregation applies only when a partial is stored. In the case of Cutty this occurs exactly $^{r}/_{s}$ times. In contrast, *pairs* writes to the store $2^{r}/_{s}$ times while in *panes* and *RA* this occurs exactly $^{r}/_{gcd(r,s)}$ and r times, as depicted in Table 5.1. Evidently, Cutty exhibits the lowest amortized cost, by invoking costly store operations half of the times compared to *Pairs*.

5.7 Experimental Evaluation

In this section, we assess the performance of Cutty compared to *Pairs* [14] and *RA* (FlatFat/B-Int)[32, 15] for periodic and non-periodic windows.

5.7.1 Experiments Setup

Implementation and Competing Approaches. We implemented the API for user-defined windows and the Cutty aggregator on Apache Flink 0.9 [43]. In order to enable a fair comparison, we implemented Pairs [14] and RA [15] within the same codebase, sharing data structures and function calls. More specifically, we implemented the following techniques[3], each of which maps to specific configurations in our framework:

- **Naive:** The *Naive* aggregator does not exploit any window semantics (e.g. periodicity) nor sharing opportunities for multiple windows. It simply recomputes overlapping windows and runs a separate aggregator instance per window query. Thus, the amount of resources utilized is proportional to the number of input queries.
- **Cutty:** The main strategy that is activated in our framework for *deterministic* window functions. Unless stated otherwise, Cutty uses an *eager* aggregate store, i.e., it maintains a tree of partials as described in subsection 5.5.4.
- **Pairs [14]:** *Pairs* is the state-of-the-art aggregate sharing technique for periodic queries. Our implementation of *Pairs* uses a *lazy* aggregate store strategy, as described in the original paper.
- **Pairs+:** An enhanced version of the *Pairs* technique that is configured with an *eager* aggregate store enabling a fair comparison against Cutty.
- **RA [15]:** The Reactive Aggregator (RA) uses a general aggregation strategy [15] that employs no slicing, relying solely on an *eager* aggregate store (FlatFAT).

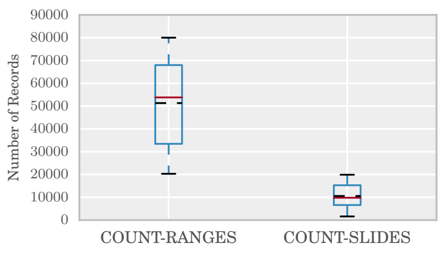
Dataset. We used the DEBS12 grand challenge dataset [95] which contains events generated by sensors of a factory. Each record of the dataset comprises of energy metrics and sensor states sampled with 100Hz rate. In total, the dataset contains roughly 33 million events. Each event includes three energy measures and 54 binary sensor-state transitions. We decided to use the DEBS12 dataset as it serves very well to generate non-periodic session-based window queries using sensor transitions as punctuations for the experiments which include deterministic/non-periodic windows (Figure 5.7.3).

Hardware. We executed all experiments on a 4.0 GHz Intel Core i7-4790K with 12GB DDR3 pre-allocated memory on a v1.8.0_91 JVM. Note that the aggregation operator in our experiments utilizes only a single core and is executed on a single task within Apache Flink's runtime.

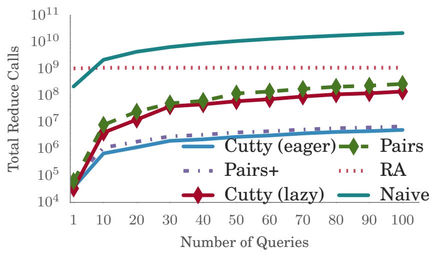
5.7.2 Periodic Window Aggregation

Workload. In this experiment we focused on periodic windows and generated a variable number of sliding count-based windows on top of the DEBS12 dataset.

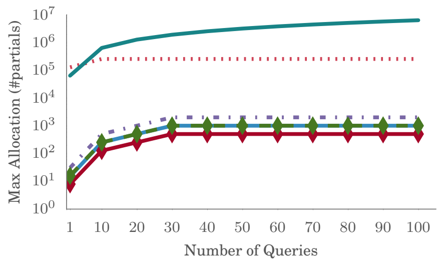
[3]The *Panes* technique was excluded because i) it is generalized and subsumed by Pairs and ii) there is no known support for multiplexing multiple sliding windows.



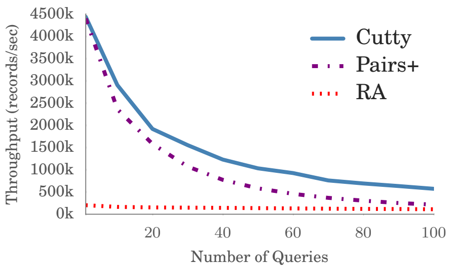
(a) Ranges and Slides of Periodic Windows.



(b) Computation Costs of all Techniques.



(c) Max Stored Partials - Throughout.



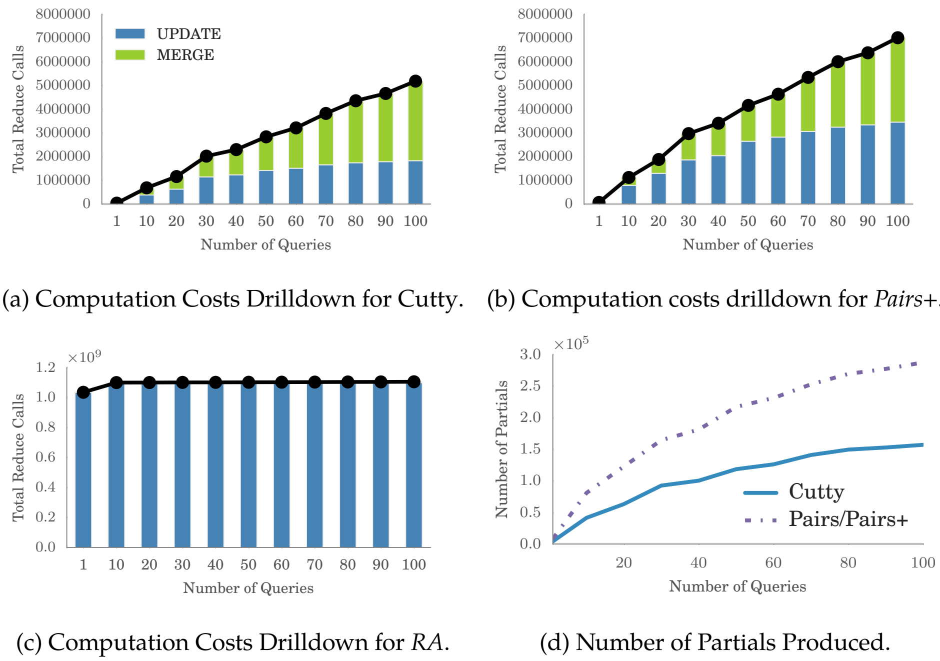
(d) Throughput of the Eager Techniques.

Figure 5.17: Performance Overview for periodic queries over workload sizes.

We used a rolling average of the three main energy measures as the aggregation function. We further generated queries of uniformly random window ranges and slides. The range values were selected within the range [20000;80000] and the slides within [1000;20000]. Furthermore, slides and ranges were chosen to be multiples of 10, thus, yielding 5000 possible unique range and 1900 slide values. The resulting distribution of ranges and slides is depicted in Figure 5.17a. In this setup, we created 11 workloads containing 1 to 100 queries.

Figure 5.17b depicts the total number of reduce calls which were executed by each of the techniques throughout the experiment (aggregating over $\sim$ 33M records) for different workload sizes (1-100). Respectively, Figure 5.17c depicts the maximum number of partials stored by each of the techniques during the experiment (i.e., maximum memory allocation). Note that the number of reduce calls correlates to the throughput and the overall performance of the techniques (we omit the graphs for the lack of space).

Benefits of Sharing. Aggregate sharing trades memory resources for less computation. A first observation in Figure 5.17b is that RA yields computational gains only when sharing more than 10 queries. This is because RA does not perform slicing and invokes a store operation per record in FlatFat. As mentioned earlier FlatFat



(a) Computation Costs Drilldown for Cutty. (b) Computation costs drilldown for *Pairs+*.

(c) Computation Costs Drilldown for *RA*. (d) Number of Partials Produced.

Figure 5.18: Performance drilldowns for periodic queries.

operations yield additional reduce calls. Besides RA, all other techniques seem beneficial to use from a single query.

Performance Comparison. Cutty with an eager strategy clearly offers the best performance results by requiring an identical amount of memory to the *Pairs* technique (the two techniques coincide in Figure 5.17c) while achieving nearly half the amount of reduce calls compared to *Pairs+* (Figure 5.17b). *RA* on the other hand, over-utilizes memory while also incurring heavy computational load, over three degrees of magnitude higher than Cutty (eager) and *Pairs+*.

Eager vs. Lazy Store. Throughout this paper Cutty assumed an eager aggregate store as described in section 5.5. The cost trade-offs depicted in Figure 5.17b and Figure 5.17c validate this decision. We can observe that both in the case of Cutty and *Pairs+* the *eager* strategy trades off exactly and constantly twice the memory requirements to gain more than an order of magnitude less overall reduce calls. The computational gain becomes more significant when sharing large amounts of overlapping window aggregates. Since the memory overhead is very small (in the orders of hundreds), for the rest of this analysis we will only focus on the performance of the eager storage strategy for Cutty and *Pairs+* (*RA* already relies on eager pre-aggregation).

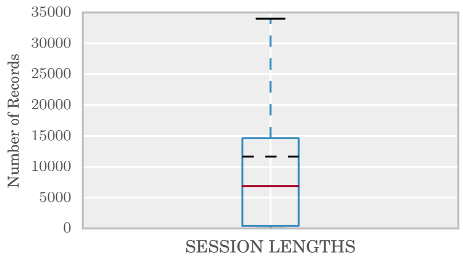
Throughput Overview. In Figure 5.17d we compare the throughput of Cutty, *Pairs+* and *RA* for the same workloads. As expected, throughput degrades for larger workloads in Cutty and *Pairs+* since more windows yield a larger amount of partials and final aggregations to execute. However, 100x more queries yield only a degradation of 5-6x for both, Cutty and *Pairs+*. In the case of *RA*, throughput is always at a minimum even for small workloads (1 or 10 queries). The reasons behind this are covered below.

Insights on Computational Overhead. To provide further insights on the performance results presented in this analysis we further decompose reduce calls occurred in the aggregator for Cutty (Figure 5.18a), *Pairs+* (Figure 5.18b) and *RA* (Figure 5.18c) for the same workloads. The calls attributed to `update` summarize the cost and frequency of the `append` operation (and associated aggregate store operations such as resizing), while the `merge` part of the calls summarizes the overall overhead caused by final window aggregation, i.e. `merge` calls. In the case of Cutty, the `merge` operations attribute to the majority of the aggregation costs, while `update` costs are kept at a minimum. On the other hand, *Pairs+* incur an almost two times higher number of pre-aggregation calls compared to Cutty with `update` operations dominating (compared to `merge` related calls) as the number of queries increases. *RA*'s operation seems to be overly dominated by pre-aggregation calls. This is because *RA* does not differentiate window specifics and eagerly stores each incoming record to its aggregation store resulting into a $\log(n)$ additional cost per record, for a significantly larger n compared to Cutty and *Pairs+*. Finally, Figure 5.18d shows that slicing in *Pairs+* results into double the overall amount of partials compared to Cutty. That clearly explains the higher computational costs of *Pairs+*, since twice more partials yield twice more frequent updates at the store.

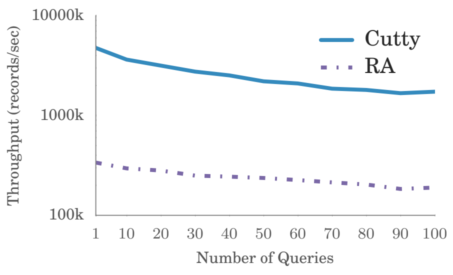
Experiment Summary. Cutty exhibits the highest throughput by at least 2x compared to *Pairs+* and nearly half the number of reduce calls compared to *Pairs+*. Furthermore, it has a computational cost at least three orders of magnitude lower than *RA*, the best known general pre-aggregation technique. The proportions of these results also align with our analytical comparison, described in section 5.6. Concluding, Cutty is the aggregate sharing technique with the least cost for periodic queries both in theory and in practice.

5.7.3 Non-Periodic Window Aggregation

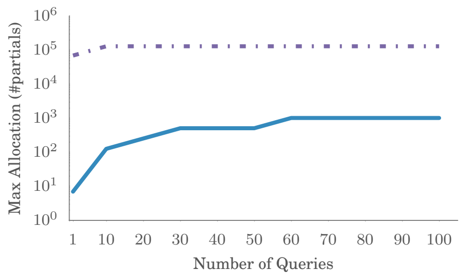
Workload. The DEBS12 dataset [95] consists of arbitrary binary sensor state transitions (true/false) that, in essence, can serve to pinpoint different sessions (i.e. 1 from 0 or 0 from 1 opens a new session). Since the actual dataset had only 20 active sensors (and 34 which seldom changed state) we implemented UDWs that simulated session windows, picked at random from the same distribution as the



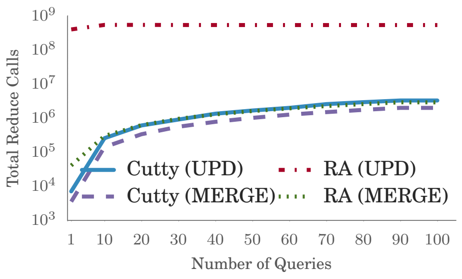
(a) Window Distribution.



(b) Throughput.



(c) Max Memory Utilized.



(d) Reduce Calls.

Figure 5.19: Computational and Memory Comparisons between Cutty and *RA* for Multiplexing Non-periodic Queries

frequency of the 20 active sensor state transitions (summarized in Figure 5.19a). This type of dynamic windows is fundamentally non-periodic (but deterministic) and thus, we only included Cutty and *RA* in this experiment.

Performance Comparison. In Figure 5.19b we can see that the throughput of *RA* is already several orders of magnitude lower than Cutty, starting from a single query. Since in this scenario the windows are not sliding, the impact of additional windows is more clear. Memory utilization (Figure 5.19c) follows similar trends that we have seen in the previous experiment with periodic queries. Additionally, in Figure 5.19d we can see a drill down of the computational costs of both Cutty and *RA*. As before, the overall computational difference is around three orders of magnitude. We can further observe though, that the number of `merge` related calls in *RA* are similar to the one of Cutty's *update* and *merge* related reduce calls. Still, `merge` calls in this scenario were significantly higher than in the previous case with periodic queries since session windows could be smaller and thus, trigger full window computations more frequently.

Experiment Summary. The performance results of Cutty for non-periodic, deterministic windows highlight the core benefit of slicing, when it is applicable. To the best of our knowledge, Cutty is the first technique that can apply slicing in non-periodic, deterministic windows while doing so with a higher efficiency and better performance than state-of-the-art slicing techniques for periodic windows. However, it is fair to note here that RA can be used in more general cases (e.g., out of order eviction of window items), not only for deterministic UDWs.

5.8 Related Work

Panes, Pairs, and RA. Pairs [14] improved upon Panes [13] by reducing the number of slices per query and by extending Panes' ideas to multi-query aggregation. However i) Pairs itself fails to scale to large numbers of queries with diverse range and slide sizes. ii) Unlike Cutty, Pairs limits the query scope to periodic windows. iii) Cutty performs better than both Pairs and Panes theoretically and experimentally . Finally, RA [32, 15] applies to non-periodic windows but with a large memory cost in order keep all individual elements of the data stream in a tree structure. In contrast, Cutty can apply slicing for the class of deterministic non-periodic windows with much less computational and memory cost.

Other Approaches. Li et al. [88, 96] did cot consider slicing techniques, but briefly classified window types by their evaluation context requirements, leaving the characterization of each class as an open research question. Our work partly fills this gap: deterministic discretization functions subsume all forward-context-free windows (no future records are required to know when a window starts), while non-deterministic discretization functions are forward-context-aware. Slider [97] showcases another example of combining partial, incremental aggregation with pre-aggregation trees for periodic windows of the same metric as well as the ability to dynamically adjust the specification of a pre-aggregation tree. However, the same approach lacks the semantic generality and adaptation of pre-aggregation schemes to multi-query computation sharing, as shown in Cutty.

5.9 Future Work

Windowing semantics are becoming increasingly complex and sophisticated the more data stream processing gets adopted. Aggregation techniques will have to follow the trends in windowing semantics and adapt to more dynamic, data-centric window types. One of the most prominent future directions in stream windowing is its standardization and encapsulation in stream SQL standards that are undergoing in open-source communities (e.g., the Calcite project [98] and Google Dataflow [21]). However, no significant efforts have been made to tight relational optimisations in stream windowing. Another future direction is to extend slicing capabilities beyond deterministic windows (if possible) and cover cases of fully data-driven windows without FIFO guarantees such as ADWIN [58]. Finally, general pre-aggregation data structures have to employ the notion of out-of-orderness, a concept not covered in our approach. Currently, with existing out-of-the-box solution such as FlatFat it is not possible to retract already evaluated window aggregates, thus, making it impossible to use for systems like Beam and Flink with out-of-order logic.

5.10 Acknowledgements

I would like to thank Jonas Traub, our MSc student assistant while working on this project, for his hard work building the first version of UDWs on Apache Flink and his help designing the evaluation framework used in this analysis. Jonas was co-supervised by Asterios Katsifodimos, Volker Markl and Seif Haridi who have contributed with great feedback and ideas during and conception and writing of this extended work. Furthermore, I would like to thank Lars Kroll, Vasiliki Kalavri and Gyula Fora for their useful feedback on many theoretical and practical aspects of this project. The Cutty aggregator blends and simplifies many interesting ideas from the domains of language semantics and database optimisation. Thus, the conception and generalization of these concepts would not be possible without many relevant discussions together with Asterios Katsifodimos, Albert Bifet, Martin Hirzel from IBM Research and engineers from the Google Compute Cloud team.

5.11 Summary: A Design Approach Perspective

Data stream windows are still one of the most central abstractions in data streaming. Aggregation techniques aim to reduce computational redundancy to the maximum extend possible for sliding windows. Most often, approaches to efficient aggregation are entangled with actual windowing semantics, such as assuming periodic queries to provide efficient pre-aggregation. Slicing techniques provide low memory footprint and generally good performance at the cost of limited applicability while general pre-aggregation techniques can be employed for any window lookup at the cost of high computational and memory footprint. To this end, we designed and implemented *Cutty*, a window aggregation framework that aims for a hybrid solution by generalizing slicing further via the use of deterministic windows while combining data structures from general pre-aggregation for an optimal result. Our general approach, adopting UDWs and employing Cutty satisfies all three of our core design principles, as follows:

[D1] Blocking-Coordination Avoidance Shared window aggregation is a problem that typically requires no distributed coordination, however, it can be a rather obstructive process if not executed efficiently, e.g., when an aggregation tree grows too big resulting into extremely slow lookups and high pre-aggregation latencies.

Our approach bounds space and computational requirements to the number of active windows which is, to our knowledge, the most latency- and space-efficient approach yet for such a general class of supported windows.

[D2] Runtime Transparency We designed Cutty in a way that it is completely transparent to the user how windows are evaluated. In fact, UDWs are a non-restrictive programming model in comparison to periodic-only windows offered by alternative approaches and systems. We believe that deterministic and non-deterministic UDWs offer a simple primitive that can serve between complex application-domain window specifications (e.g., trigger and eviction policies in IBM Stream Processing Language [57] and sessions in Apache Beam/Cloud Dataflow [21]). Our prototype of Cutty could trivially translate most known window templates such as periodic windows on different measures as well as session, tumbling and landmark windows into UDWs at compilation time.

[D3] Model Compositionality UDWs enable programmers to assemble a complex cross-window aggregation without the need to find and tweak slicing or other execution-centric parameters. For example, window slides in the original work of Discretized Streams [12] would optimally need to be multiples of the microbatch windowing, expressed in real time. Such restrictions are avoided with the use of UDWs since the application semantics can be totally decoupled from the aggregation needs. Furthermore, as analyzed already in D2, UDWs can be composed by and also compose other UDWs. Most importantly, Cutty is to our knowledge the first aggregator that allows fusion of sliding windows on heterogeneous measures such as count and time-based windows, sharing the same workload efficiently.

CHAPTER

6

Iterative Data Streaming

6.1 Introduction

So far throughout this work we have addressed two broad challenges in stream processing: state management and efficient sliding window aggregation. A stream processor with persistent managed state and the notion of window aggregation can in combination support reliable and scalable deployments of relational operations, complex event processing tasks and generally any computation achievable within a single pass of the data. However, there exist computational problems that require multiple passes to extract deeper metrics and relations hidden in the data. Typical examples can be found in the fields of graph analytics, machine learning and logic programming all of which are emerging to become prevalent within modern data management workloads.

Iterative processing has been supported inherently well for batch processing systems, using their built-in centrally coordinated execution and the use of external shared state in distributed file systems[17, 99, 100, 101]. However, enabling multi-pass computation in stream processing introduces several interesting challenges that have to be addressed while taking into consideration the existing design choices that are already in use such as coordination-free task execution, in-flight asynchronous snapshotting and flow control. To that end, we identify a set of runtime additions as well as programming model primitives that are essential to iterative processing, yet, current production-grade stream processors fail to provide.

At the runtime level, there is a need to compose arbitrary nested dataflow computation. That means that we should be able to incorporate concrete loop structures into stream process graphs and identify their scopes explicitly in order to be able to determine and track concretely how a multi-step, distributed computation progresses and when it terminates. Having additional levels of encapsulation in process graphs brings further challenges such as the ability to associate individual records to their corresponding computational scopes and a scalable way to collect, track and notify the progress of each cyclic computation to its underlying distributed tasks while avoiding central coordination and deadlocks due to network congestion within cyclic subcomponents. Important challenges also arise at the programming level in order to offer convenient semantics for users to express cyclic computation that is nested within a continuous pipeline. In essence, we seek for ways to bring well-known iterative processing primitives from prior specialized systems such as bulk or stale superstep synchronization and fix-point termination in the context of streaming. The user-facing model should be compositional and integrate well with managed state and the existing single-pass aggregations (e.g., on stream windows).

Previous approaches that consider iterations on data streams are either overspecialized to a specific domain (e.g., Graph analysis [102, 103, 104]) or computational problem at hand (e.g., Flying Fixpoint for reachability in Datalog [105]). The only exception is the Timely Dataflow model [106] implemented by the Naiad system [20] which adopts an asynchronous gather and scatter model for observing iterative and other progress hierarchically via the use of intermediate tracking processes. While Timely Dataflow's approach is a good example of a general approach it does not integrate well with asynchronous epoch commits and in-flight flow control which requires all computational logic and state to remain in the stream process graph.

We propose an extension of single-pass stream processing that offers complete support for arbitrary nested iterations and builds on top of low watermarking [22], the dominant progress communication mechanism in production dataflow systems (e.g., Apache Flink, Beam [42, 9], Spark's Structured Streaming [107], Kafka Streams [108] and Storm [38]). We investigate how scopes can be used in combination with Progress Timestamps, an extension of low watermarks for nested monotonic measures of progress to invoke tasks about different types of progress updates. We further implement a window multi-pass aggregation operator that showcases the capabilities of the system to multiplex iterations across different stream windows while giving access to underlying persistent and per-window managed state. Finally, we showcase the applicability of window multi-pass aggregations to provide a graph-centric programming model for dynamic graphs and show how we can implement standard algorithms such as PageRank, Connected Components and Single Source Shortest Paths (SSSP).

The outline of this chapter goes as follows: section 6.2 describes a model for iterative processes step-by-step from its basic representation to distributed execution semantics and bulk synchronous processing in different execution environments. Then, section 6.3 describes all proposed underlying stream model abstractions needed to integrate iterative processes in a long-running stream execution of independent tasks. In section 6.4 we describe a prototype window aggregation library that makes use of the proposed stream model abstractions to implement multi-pass window aggregations. Finally, section 6.6 offers all the acknowledgements specific to this work, followed by an extensive summary in section 6.7 that discusses the use of our core design principles in this proposed solution.

6.1.1 Contributions

We can summarize in more detail all of the contributions of this work into the following:

- We identify a small universal set of programming primitives that describe an iterative process.
- We explain existing systems and approaches for iterative processing using our primitives.
- We present the problem of iterative processing as a special case of out-of-order processing in data streaming.
- We provide a complete model design that allows fully decentralized cyclic,out-of-order progress tracking and the composition of arbitrary nested cyclic computation.
- We showcase the usages of our model with an implementation of a multi-pass window aggregation framework on Apache Flink.

6.2 Preliminaries

Before we begin our study for iterative processing semantics in data streaming we will first describe the iterative process as a simple programming abstraction and examine its core structural properties that we wish to encapsulate in data streaming.

6.2.1 A Basic Model for Iterative Processes

The concept of an iteration is a standard building block for many algorithms in domains of computational statistics, machine learning, graph and logic programming. It typically consists of a block of operations that repeats, generating an updated solution each time, until a condition is satisfied. The iterative process bears similarities to general cyclic control flows in programming languages, yet, it is a higher level abstraction. For example, an iterative process would be used to minimize a differentiable objective function or approximate the location of cluster centroids in recurrent steps which consider full access to the data or state at hand on each step. This is different than enumerating the elements of a list or modifying an array in a loop. In this section, we will present a simple model for composing iterative processes starting from a non-parallel execution model and later extending it for distributed processing.

6.2.1.1 Loops, Nesting and Termination criteria

An iteration can be typically decomposed into two functions, a *loop* function and a *termination* function. The *loop* function $L : X \rightarrow X$ encapsulates the block of repeated operations on a given state or "solution" of type X while the *termination* function $c : (X, X) \rightarrow \text{bool}$ yields `true` when the termination condition is satisfied. Given these two primitives and an initial state $x_{init} \in X$ a iterative computation could be composed as follows:

```
def iterate(c,L)(x_init)
    x = x_init
    do{
        x = L(x)
    }while(¬c(x,L(x)))
    x
```

The above composition of an iterative process suffices when the termination criterion and loop function are only dependent on the state of the computation. For example, a fix-point iteration that terminates when the solution converges (i.e., no changes in a step) could be defined via the following termination criterion based on the model above.

```
def c(x,x′) = x==x′
```

Other variants of fix-point computation such as error based convergence criterion based on a given error ϵ could be similarly defined as such:

```
def c(ϵ)(x,x′) = |x−x′| ≤ ϵ
```

Furthermore, our definition above also supports composition via the use of partial application. More concretely, the loop function can itself be a nested iteration (at arbitrary levels) with a termination condition c' and loop function L':

```
val x = iterate(c,iterate(c′,L′))(x_init)
```

6.2.1.2 Identifying Iterative Progress

Despite the simplicity of our model, there are cases when the loop computation or the actual termination condition need to keep track of the current computational step. A typical way to encapsulate this notion of progress in a chain of consecutive iterative steps: $L^0(x), L^1(x), \dots, L^p(x)$ is via a step counter $p \in \mathbb{N}$ that acts as control variable accessible within the context of an iteration. In such a model we can support a progress-aware loop function $L : (\mathbb{N}, X) \rightarrow X$ as well as a termination condition $c : (\mathbb{N}, X, X) \rightarrow \text{bool}$ within an iteration expressed as follows:

Operation	**Implem.**	**Example**
progress: $\mathbb{N}^n \rightarrow \mathbb{N}$	head(P)	p_n
unnest: $\mathbb{N}^n \rightarrow \mathbb{N}^{n-1}$	tail(P)	$[p_{n-1}, \dots, p_1]$
nest: $\mathbb{N}^n \rightarrow \mathbb{N}^{n+1}$	0 :: P	$[0, p_n, \dots, t_1]$
incr: $\mathbb{N}^n \rightarrow \mathbb{N}^n$	head(P)+1 :: tail(P)	$[p_n{+}1, \dots, p_1]$

Table 6.1: An Overview of Operations Supported by Progress Timestamps

```
def iterate(c,L)(x_init)
    (p,x) = (0,x_init)
    do{
        (p,x) = (inc(p), L(p,x))
    }while(¬c(p,x,L(x)))
    x
```

This model suffices to describe all common types of iterative processing. For example, a typical fixed iteration process with k steps could be implemented by the following termination condition that is satisfied when the loop counter is higher or equal to k:

```
def c(k)(p,_,_) = p >= k
```

Example: As a concrete example, suppose that we want to approximate the value of π (pi) using an iterative process. With the primitives above we can express an iterative algorithm (Borwein 87) which terminates when a converging criterion is met or, at the worst case, when a maximum count of steps has been reached.

```
def piLoop((p, (x_0, y_0, pi_0)))
    x = ((sqrt(x_0) + 1)/sqrt(x_0))/2
    y = ((1 + y_0)sqrt(x_0))/(x + y)
    pi = (((1 + x)pi_0)y)/(1 + y)
    (x,y,p)

def piTerm(maxSteps, ϵ)(p, (_,_,pi_n), (_,_,pi_n+1))
    (p ≥ maxSteps) || |pi_n+1 − pi_n| ≤ ϵ

> iterate(piTerm(10^10, 10^−20),piLoop)(sqrt(2), 0, 2 + sqrt(2))
```

If we now try to apply nesting in progress-aware iterations we face an interesting challenge, that of passing and identifying outer-scope progress in nested processes. That is generally important since the progress of a nested iteration is not only a local step counter but a metric that captures all encapsulating iterative processes hierarchically that led computation up to this distinct step. As we will cover later in this chapter, this information is especially important when the cyclic computation is executed in a data-parallel fashion across different machines.

Progress Timestamps: A simple extension we can make to the basic iteration model in order to encapsulate progress is to introduce a *Progress Timestamp* type that is a list $P \in \mathbb{N}^n$ of loop counters at n nested levels as such $P = [p_n, p_{n-1}, \dots, p_1]$, where p_n corresponds to the current iteration step at nested level $n \in \mathbb{N}$. Progress timestamps can support the following four basic operations: *progress, unnest, nest and incr* as summarized in Table 6.1. With progress timestamps and their operations we can trivially go ahead and extend our basic iteration model with arbitrary nested progress support.

```
def iterate(c,L)($P_{init}$,$x_{init}$)
    (P,x) = ($P_{init}$.nest,$x_{init}$)
    do{
        (P,x) = (P.incr, L({P,x))
    }while($\neg$c(P,x,L(P,x)))
    x
```

Given the model above, we can make both the termination condition and loop function completely aware of the local progress, i.e., via $P.progress$, as well as the progress of all encapsulating levels, i.e., $(P.unnest)^k.progress$ for the encapsulating iteration process at k levels.

Example: To motivate the expressive power of our basic iterative process model with nested progress measures, consider the highly nested iterative computation of the radix-2 Fast Fourier Transformation (FFT) algorithm [109]. Radix-2 FFT divides a discrete fourier transform of size N into two interleaved processes transforming $N/2$ values on each global superstep, also known as butterfly steps. For brevity, we express only the core iterative steps of the algorithm (the bit reversal permutation is trivially implemented by the inverseBit function at the input).

```
def KTerm(P,Y,_) = $2^{P.progress+1} > Y.size$
def TTerm(P,Y,_) = $P.progress > Y.size - 2^{P.unnest.progress+1}$
def JTerm(P,Y,_) = $P.progress > (2^{P.unnest.unnest.progress+1}/2) - 1$

def Radix2FFT(P,Y)
    K = $2^{P.unnest.unnest.progress+1}$
    t = $P.unnest.progress$
    j = $P.progress$
    //$w_N$ is a shorthand for $e^{\frac{2\pi}{N}}$
    $\begin{pmatrix} y_{t+j} \\ y_{t+j+K/2} \end{pmatrix} \leftarrow \begin{pmatrix} y_{t+j} + w_K^j \cdot y_{t+j+K/2} \\ y_{t+j} - w_K^j \cdot y_{t+j+K/2} \end{pmatrix}$

> iterate(KTerm,
>   iterate(TTerm,
>      iterate(JTerm, Radix2FFT)))(nil, inverseBit(Y))
```

6.2.2 Bulk Synchronous Parallel Iteration

So far, we have derived a powerful set of computational attributes for iterative processing, namely termination(c), looped computation (L) and state (X), however, we have not addressed yet the possibility of distributing its execution. In fact, the marriage of data-parallel computing and iterative processing is an especially attractive combination, since it can be used to solve approximations that are only possible at large scale when state and computation are partitioned.

Data parallel batch compute models, such as MapReduce [8] and its corresponding system implementations seen in Hadoop[110] and Spark[99] all have something important in common, they are embodiments of Valiant's Bulk Synchronous Processing (BSP [16]) bridging model for distributed computing. According to the BSP model distributed computation is organized into concrete supersteps (parallel steps) and consists of the following three basic elements : 1) distributed components/processes executing local instructions, mutating local state if any, 2) a router abstraction R that routes messages between any two components and 3) a synchronization primitive (sync()) which allows progress only when a superstep has been completed across all components.

We can update our iterate process model to an *iterateBSP* model by simply introducing R, the router abstraction and a $sync$ operation that blocks local execution until the global completion of a superstep, including the routing of outstanding messages. We assume that R supports a $send(M)$ operation that routes a set of output messages M, addressed by their corresponding target component, as well as a $receive()$ operation which yields M, a set of all incoming messages at the target component. A BSP-compliant iteration process would simply have to block until a superstep is complete and then execute a local computation $L : (P, M, X) \rightarrow (M, X)$ that incorporates input messages received in the current superstep with the output messages to be delivered in the next superstep.

```
def iterateBSP(c,L)(P_init, x_init)
   (P,M,x) = (P_init.nest, ∅, x_init)
   do{
      (P,(M,x)) = (P.incr, L(P, R.receive(), x))
      R.send(M)
      sync() //blocks until global computation and routing are complete
   }while(¬c(P,x,L(x)))
   x
```

The BSP-compliant iterative model presented above has sufficient expressive power to be used in sophisticated compute problems that require large scale such as distributed graph algorithms and stochastic gradient descent, yet, it is simple enough to be implemented by different underlying distributed execution engines. For better understanding we will do a small recap of some of its existing usages as well as how it has been implemented on top of existing systems.

6.2.2.1 Iterations in Distributed Graph Processing

Graphs are a dominant data model for expressing relations of complex nature such as people's interactions in a social network, cross-references on scientific work or any web content as well as dynamic events occurring in the physical world such as road traffic and virus outbreaks. The arbitrary complex interconnections of vertices further make the definition and data-parallel execution of graph algorithms hard due to the high amount of data-dependencies. In the recent years, the vertex-centric programming model emerged to become one of the dominant ways to express and execute graph computation. Google's Pregel [17] system first introduced the notion of vertex centric computing as a special application of BSP [16] where distributed components represent vertices in a graph which can send and receive messages to their neighbors via strict synchronization BSP rounds. The same model was adopted by other popular graph processing libraries such as GraphX [111] built on Apache Spark [27]. Our general iterate BSP process model fully subsumes the vertex-centric programming model via its use of local state, routing, synchronization and termination condition. In the example below, we show how we can implement the iterative PageRank [112] algorithm with fixed iterative termination and the trivial addition of the library-specific static calls $getNeighborIDs()$ and $vertexCount()$ that yield a set of IDs in the neighborhood of a vertex and the number of all vertices in the graph respectively.

Listing 6.1: Page Rank as an Iterative BSP Process

```
def pageRank(P,M_IN,x)
   weightSum = M_IN.map(m => m.value).sum
   x = 0.15 / vertexCount() + 0.85 * sum
   outDegree = getNeighborIDs().count
   M_OUT = getNeighborIDs().map(id => msg(id,x/outDegree))
   (M_OUT,x)

def pageRankTerm(supersteps)(P,_,_) = P.progress > supersteps

> iterateBSP(pageRankTerm(40),pageRank)(nil,Graph)
```

Notice that while iterative computation is local, per vertex, the input and output of the overall iterative process is the collection of all approximated vertex states (i.e., ranks) in the graph. In the example above, we operate on a full graph of ranks, though, in systems like Spark and Flink these are higher order representations that are backed by a distributed data type such as Spark's Resilient Distributed Dataset (RDD) or Flink's DataStream respectively. The details as to how a distributed data type (e.g., all ranks in the partitioned graph) is being partitioned and how iterative computation is scheduled is merely a system-specific implementation concern and thus, we choose to omit in this part for generality.

Message-Based Termination: Given that in distributed computing systems such as Pregel [17] the state is partitioned, it is often easier to infer fixpoint termination based on the messages exchanged per round instead of inspecting all state changes on individual partitions. Messages directly correspond to state updates for a large number of BSP algorithms such as Connected Components. In Listing 6.2 we present an adaptation of the Connected Components algorithm to our iterate BSP model that uses an alternative termination condition on the number of messages instead of the state change.

Listing 6.2: Connected Components as a Fixpoint Iterative BSP Process

```
def connectedComponents(P,M_IN,x)
   componentID = M_IN.min
   if (x != componentID){
      x = componentID
      M_OUT = getNeighborIDs().map(id => msg(id,x))
   }
   (M_OUT,x)

def ccTerm(P,M) = (M = ∅)

> iterateBSP(ccTerm,connectedComponents)(nil,Graph)
```

The iterative connected components algorithm, as defined in the Pregel [17] framework, can be declared in an iterative BSP process that terminates when a fixpoint has been reached. The algorithm takes as an input a distributed graph of vertices having unique identifiers and their list of neighbors. Initially, every vertex uses its own identifier set as its connected component. On each iteration step, a vertex selects the minimum id over the received messages as its connected component and further broadcasts it to its neighbors if it has changed. The iterative process proceeds until no messages are exchanged any more in an iteration step.

6.2.2.2 SGD and Bounded Asynchrony

Another domain of iterative processing that has seen wide adoption in the recent past is machine learning. Stochastic Gradient Descent [113] (SGD) and its parallel incremental implementation [114] is the most popular method today for optimizing differentiable objective functions. One of the distinct properties of SGD and generally the iterative evaluation of convex functions, is their ability to converge faster while maintaining accuracy at relaxed distributed synchrony models [19]. The stale synchronous parallel model, one of the dominant synchronization methods, allows tasks to proceed with their computation independently up to a bounded degree of asynchrony i.e., $m \in \mathbb{N}$ number of steps ahead from the slowest task in the system, known as *slack*. This allows better utilization and convergence rates at predictable accuracy losses.

Our iterate BSP programming model can allow bounded asynchrony given a modification to the synchronization primitive that considers a slack parameter $s \in \mathbb{N}$ such that sync(s) allows progress when other parallel iterative processes lag at most s supersteps behind the currently completed superstep. For brevity, we omit a re-definition of the progress function, yet, we will consider the ability to express and execute bounded asynchrony in our proposed model presented in section 6.3.

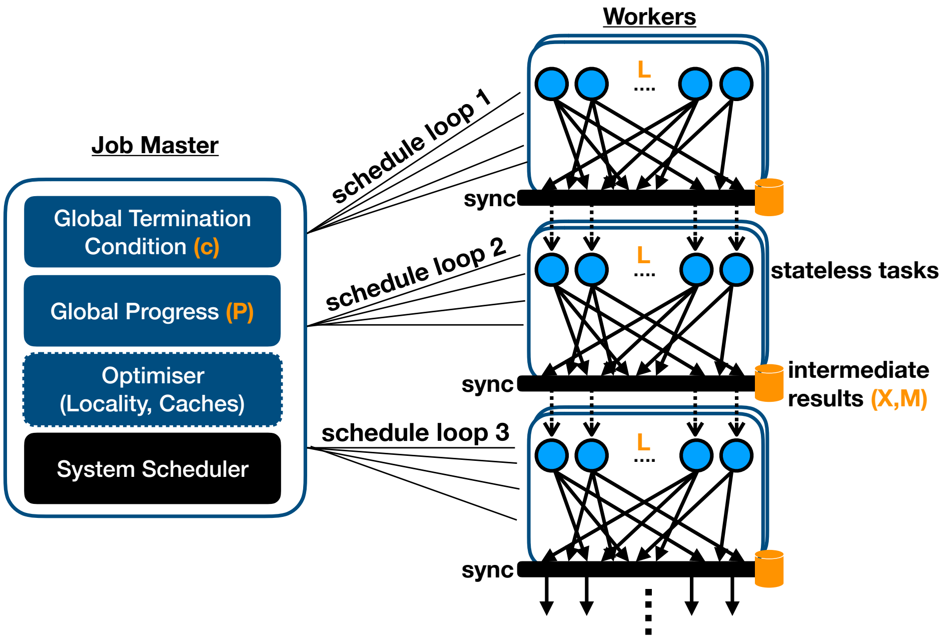


Figure 6.1: Overview of strictly coordinated iterative processes on batch compute systems.

6.2.3 Short-Lived Task Execution of Iterative Processes

The implementation of bulk synchronous parallel iteration models on existing short-lived distributed computing systems has been an active research topic of the past decade, primarily in the context of the popular MapReduce [8] architecture. Existing approaches augment the existing centrally coordinated, staged task execution of MapReduce-based systems (map, shuffle and reduce operations) with iterative process semantics. More concretely, systems such as Haloop [100], Twister[115] as well as Spark [27] showcase that an iterative process can be unrolled into a series of stages. In Figure 6.1 we summarize the core ideas, where each superstep of an iterative process involves scheduling a set of stateless tasks to execute it, while the tracking of iterative progress and termination conditions are evaluated globally at the application level, within the job master process. Each of the iterative process primitives we defined before can be therefore supported as follows:

Scheduling Loop Computation (L) and Synchronization The loop computation is typically scheduled to run in a dedicated Map-Reduce stage [100, 115] per iteration and more specifically within a *reduce* phase followed by a *map* and *shuffle* phase to route the messages that will be used in the next round. Bulk synchronization is inherently supported by the transactional processing of consecutive stages, both in MapReduce extensions and Spark jobs. That is due to the fact that synchronization requires all messages to be routed and computation to be finished before executing the next superstep. In all cases, a superstep is scheduled only after the completion of the previous one (i.e., map, shuffle and reduce phases).

Message Routing and State (M,X): Message routing requires a set of generated messages to be routed to the next superstep instance of an iteration. The *shuffle* phase of MapReduce can sufficiently implement that functionality since it involves routing and grouping messages by key. Similarly, the iteration state (if supported) will also have to be committed to a distributed file system or to be replicated in-memory (in the case of Spark) in order to be used in the next iteration superstep, since scheduled tasks in all batch execution models are inherently stateless. This introduces extra communication complexity, however, in most approaches [100, 115, 27] a central optimizer can make sure that data locality is kept to maximum by re-scheduling tasks across iterations in the same physical partition to access the same data (on Spark this is explicitly achieved through the cache() operation). Furthermore, for partitions with invariant state across iterations (e.g., in fixpoint iterations) caching [100] can be employed to avoid loading and shuffling it unnecessarily.

Termination Condition and Progress (C, P): Typically, a progress tracking component is typically invoked when a stage has completed, it then increments a global superstep counter and invokes a global termination condition check. For fixed iterations the condition is trivially based on the loop counter, however, for fixpoint termination the condition has to consider the actual data. A decentralized but costly approach would be to add another map/reduce step to compare the states of the last two iterations. Instead, Haloop [100] proposes the use of caching and indexing of the local output of the reducers. This approach avoids the need for a dedicated map-reduce step to implement fixpoint termination checking since the checking can be done directly at the reducers that complete an iteration result. Finally, in the end all reduce tasks report their decisions to the job master node via asynchronous communication and final scheduling decision is made at the master node.

6.2.3.1 Discussion

There have been many optimisations proposed to boost the performance of iterative processes on top of short-lived distributed task execution. However, we identify several critical limitations. While several systems might address one of these limitations no approach to our knowledge deals with all of them with a universal system design. The first limitation is the ability to provide arbitrary control flows such as scheduling nested iteration processes. That is often the case due to the already high degree of context switching needed to pass (or cache) state and messages from one stage to the next. The introduction of nested iteration states and messages would have to include all data from the encapsulating iteration processes which makes the adoption of nested structures complicated and inefficient. Second, progress and termination criteria are centralized. This is a fundamental limitation of short-lived task execution frameworks, given that tasks do not have the capability to maintain a persistent view of an execution alongside progress metrics and arbitrarily mutable state to take local decisions, thus, central coordination and bookkeeping are unavoidable in many cases. This limitation can be observed in Ciel [116] which claims arbitrary control flow execution on top of short-lived tasks, however, it adopts a centrally coordinated architecture for bookeeping dynamic object dependencies and progress. Finally, it is hard to implement relaxed synchronization primitives such as stale synchronous iterations on top of a batch processing system, unless changing its internal mechanisms completely. Having strict execution stages in all of these systems, including Apache Spark, makes the mapping of a global superstep to one stage a necessary inconvenience.

6.2.4 Long-Lived Task Execution of Iterative Processes

The concept of long-lived task execution is tightly connected to stream processing and can be summarized as the notion of deploying and running a graph of tasks that execute all stages of a computation instead of scheduling each set of tasks in individual stages. This approach in iterative processing has been generally adopted in computationally-intensive domains of data analysis since it can yield better performance through the use of asynchrony and persistent local state (i.e., no context-switching). Major efforts to provide a continuous iterative execution have been made within respective application domains. In graph processing, systems such as Kineograph [103], GraphLab [117] and Chronos [104] facilitate the execution of iterative graph algorithms with strong locality requirements. Likewise, similar efforts have been made in the field of machine learning and statistical approximation on multiplexing iterative processes and incremental updates over long-lived task execution. Tornado [118] is a an example of a system that can share approximate results of a stochastic gradient descent (SGF) across different long-running iterative approximations. While all these approaches solve certain iterative problems, they lack generality when no certain assumptions can be made when it comes to termination (convergence) criteria and loop function properties (e.g., whether it is convex). A few noteworthy efforts [10, 20, 106, 119, 105] to provide a more general programming and execution model for iterations for long-running tasks share a few common characteristics, depicted in Figure 6.2, which we summarize below.

Loop Computation, State and Termination (L, X): Computation and state in long-running execution are tightly integrated. As we already covered in the context of our core stream process model in chapter 3(Section 3.3.1.1), each task invocation

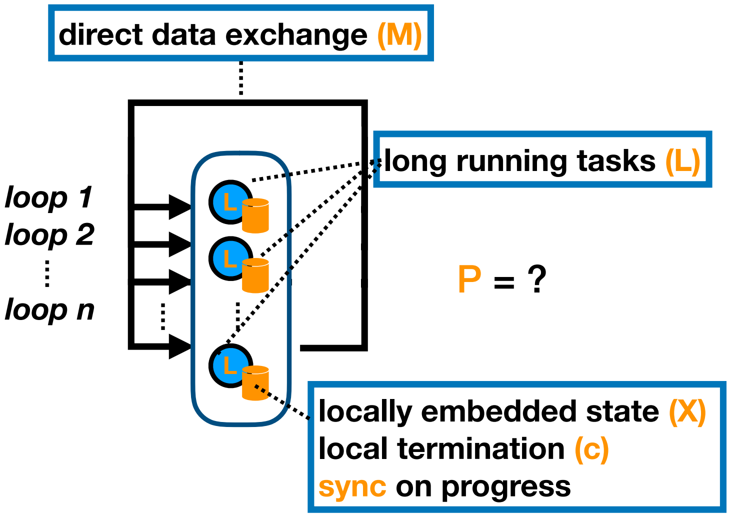


Figure 6.2: Overview of iterative processing on long-running tasks.

is message-driven and leads to generated output and a state transition. This is a natural model for implementing iterative processes, given a partitioned state X and its incremental transformations occured across iteration steps. Tasks are interconnected and loops are, in fact, actual data channels that feed messages back in a stream processing topology (in SEEP[10] loops can also be locally evaluated implicitly when no parallel data dependencies are involved in a computation, however, iterative process primitives and task synchronization are not the main focus of that work). Given that messages are shuffled during the execution across loops (compared to being stored or cached in short-lived task execution), it is possible to embed special logic (e.g., local operators) that invokes a termination condition based on the data exchanged across iterations [119].

Message Routing (M): An important distinction to short-lived execution is that messages (M) are continuously produced and consumed. As we have seen so far, data dependencies in a stream process graph correspond to a *physical* network configuration where network channels connect different parts of the computation. This yields a great advantage for iterative processing, allowing less synchronous, pipelined processing of the loop logic which can facilitate stale synchronous execution, a paradigm that was otherwise hard to implement in short-lived execution systems.

Synchronization and Progress (P): The remaining and most challenging requirement in asynchronous long-lived execution is the ability to track the progress of

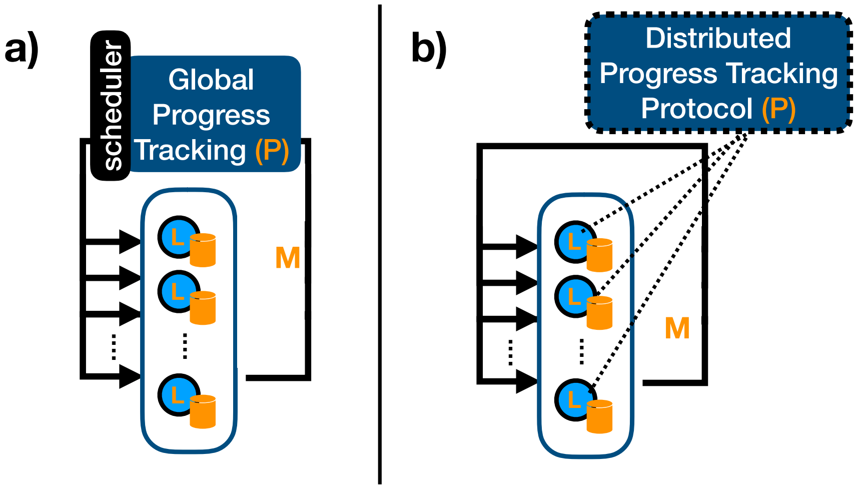

a)
scheduler
Global Progress Tracking (P)
L
M
b)
Distributed Progress Tracking Protocol (P)
L
M

Figure 6.3: Global and Decentralized Progress Tracking.

parallel subtasks and infer when a computation is complete. The common lack of a central coordinator further makes this problem non-trivial since progress needs to be typically approximated in a decentralized fashion without any global knowledge. This problem, which we informally call "progress tracking" subsumes the need for a distributed synchronization mechanism, since it can grant the knowledge of when a computational step is complete at the level of each individual task. Progress tracking is covered thoroughly in section 6.3 and, as we will show later in the chapter, is sufficient to implement more complex synchronization mechanisms on top.

We distinguish two relevant approaches to tracking distributed progress for long-running task synchronization (depicted in Figure 6.3): a) Global and b) Decentralized Progress Tracking which we summarize next.

6.2.4.1 Global Progress Tracking

Global progress tracking is known from the Naiad [20] system and its Timely Dataflow [106] model. Naiad provides programming primitives and a runtime architecture for supporting arbitrary nested iterations in a long-running task execution. Its execution model has some noteworthy differences compared to our stream process model presented in chapter 3. First, all message exchange is controlled by a global (per-machine) message scheduling unit which intercepts communication and updates a shared data data structure of progress metrics (based on pending record counts per unique timestamp) which we can call the 'tracker'. The tracker's role is to maintain an eventually consistent notion of a "clock" which, at any time, can give an approximation of the computational progress (known as the frontier) at all parts of the stream process graph. A local scheduler can then schedule notification events to tasks based on that progress and allow the emulation of synchronization barriers. Global progress tracking has shown to work well given coarse grained metrics such as computational epochs and iteration steps. However, its adoption to a fully decentralized stream process graph execution (as discussed in chapter 3) would be unrealistic given the latter's task-local independent execution as well as the need to incorporate existing decentralized mechanisms such as ultra fine-grained event time windows with watermarks [22] and marker-based snapshotting [29, 30].

6.2.4.2 Decentralized Progress Tracking

Little effort has been put into tracking iterative progress of long-running tasks in a decentralized fashion. The main idea of such an approach is to allow computational tasks to infer the distributed progress of an iteration (e.g., supersteps finished) simply by using the existing message exchange facility in stream process graph, without any external circumvention. Such an approach can be promising for wider adoption of iterative stream processing on existing production-scale platforms which consider no external communication or control unit in an execution. Among existing studies, the Flying Fixpoint (FFP) [105] approach shows promising early results of adopting an in-flight protocol for progress tracking, in the context of supporting recursive Datalog queries on a stream processing execution. Furthermore, Ewen et al. [119] also examine the prospect of inferring and detecting change in a distributed iterative process using inflight messages. Whatsoever, no general approaches are known to model and tackle the general problem using purely local computation.

6.2.4.3 Discussion and Open Problems

Long-running task execution systems are highly attractive for implementing iterative processes. While many of the basic components of an iteration (local state, message exchange etc.) have been trivially defined in a distributed stream processing setting, the problem of decentralizing progress tracking and synchronization mechanisms still remains open. Perhaps one of the main reasons we have not yet seen a clear design yet is the inability of existing decentralized stream mechanisms (e.g., low watermarking [22, 9, 23]) to operate on stream process graphs with arbitrary cycles. In 6.3, we examine these limitations as well as the potential of re-using existing mechanisms present in most distributed stream processing systems today, in order to offer the fundamental abstractions for building iterative processes in a continuous execution. Then, in section 6.4 we present an actual framework exposing iterative processes as a special case of window aggregations which builds on these abstractions.

6.3 A Model for Out-of-Order Iterative Processing

We want to enable support for arbitrary nested iterative processing on deployable stream process graphs of long-running stateful tasks. More concretely, we aim to implement all of the iterative process semantics we introduced in section 6.2 on top of the general stream process model presented in Section 3.3.1.1. Ensuring a blocking-coordination-free, fully decentralized iterative execution is the main design principle of our approach and a general requirement of our core stream process model presented earlier in chapter 3.

Our approach builds on the notion of out-of-order (OOP) processing, used for progress-based blocking computation. We identify iterative processing as a special case of OOP and provide a complete, decentralized solution to integrate cyclic progress measures to the OOP scheme.

6.3.1 Out-of-order Stream Processing

As shown previously in Section 3.3.1.1, computation in stream process graphs can lead to different distributed executions, since tasks operate independently from each other, pulling records across their incoming channels and processing them without conforming to any total ordering guarantees. On one hand, this makes stream processing a fast, coordination-free architecture for low latency processing. On the other hand, it requires potential synchronization mechanisms to be built on top of out-of-order record delivery across different input streams.

Out-of-order processing [22, 23] (OOP), is a processing paradigm that builds on top of the default (asynchronous) stream process model to offer progress and synchronization primitives. The original need for OOP was the lack of a consistent mechanism to reason about total processing order for time-based computation (e.g., application time windows) in a stream execution. Google's dataflow model [21], currently fleshed out within the Apache Beam [42] framework popularized the notion of OOP which was later adopted by Flink's programming model among others and implemented via the use of low-watermarking [22, 9]. In this section, we will explain the OOP model with an emphasis on low watermarking, a decentralized progress tracking mechanism for OOP that is currently restricted to acyclic stream process graphs but can be extended in order to support iterative processing as we will show later.

6.3.1.1 The Notion of Progress Metrics

Despite the fact that stream tasks are being continuously invoked per input record, the logic that is being executed within these tasks (which we will refer to as the "operator") can be categorized as *blocking* or *non-blocking*. *Non-Blocking* operators are those whose logic is not order-sensitive. For example, a simple *filter* or *union* are stream operators that invoke a local computation and produce an independent output per input event.

On the other hand, there exists a class of *blocking* operators whose completion is conditioned on a total order measure. For example, a stream window on application time would complete once all records up to a specific time are guaranteed to be processed. Given the absence of an atomic "application" clock, it is impossible to access that precise measure by each individual physical task at any time. In fact, a total processing order on that measure would be infeasible without a strictly coordinated execution. Instead, the out-of-order processing paradigm, as the name suggests, allows unordered processing while using a "progress metric[1]", a weaker notion than that of total processing order but sufficient for the needs of blocking stream operators. In Definition 6.3.1 we present the notion of a progress metric for a pre-defined total order of records in a stream processing execution, defined in relation to our core stream process model (see Section 3.3.1.1).

Definition 6.3.1. Progress Metric: Let R be a data stream and *event time* **ts** a total order relation $\mathbf{ts} : R \to \mathbb{N}$. Given a system transition $(s_p^i, m^i, M, s_p^{i+1})$ by a stream task $p \in \Pi$ with a state domain $\mathbf{S} : s_p, s_p' \in \mathbf{S}$, a progress metric relation $T : \mathbf{S} \to \mathbb{N}$ satisfies the following:

- $T(s_p^i) \leq \mathbf{ts}(m^i)$
- $T(s_p^i) \leq T(s_p^{i+1})$ (monotonicity)

The gist of progress metrics lies at the observation that different blocking operations can be asynchronously invoked by records arriving out-of-order (i.e., when $\mathbf{ts}(m^i) > \mathbf{ts}(m^{i+1})$). However, certain operator logic solely requires a monotonic metric to complete. In distributed system terms progress metrics provide a *stable property*[2] [120] that is known to be satisfied at a certain time in an execution. Given that streams can be arbitrarily unordered even before they are ingested by a stream processing system, a user-defined constraint needs to be set on the *maximum expected degree of unorderness* or *time slack*, which also influences how long the system has to wait until it issues updates to its progress metric.

6.3.1.2 Progress Tracking via Low Watermarking

We refer to "progress tracking" as an underlying mechanism that implements a progress metric solely by observing and cross-relating record timestamps in a stream processing system. Low watermarking [22] is a widely-used mechanism today for tracking distributed event-time progress. The main intuition behind low watermarking is similar to our snapshotting algorithm presented in chapter 3, that

[1]The original paper[22] makes use of the term *progressing attribute*

[2]A condition that cannot be violated for the rest of a distributed execution.

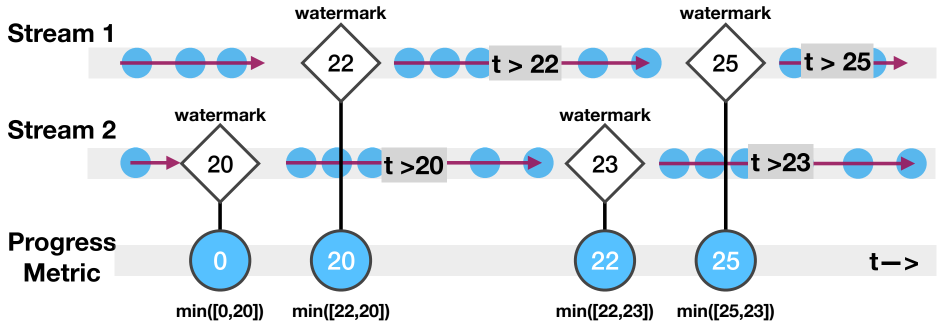


Figure 6.4: A depiction of observed low watermarks and a derived progress metric.

is "Given an *acyclic*, connected stream process graph we can disseminate updates to a progress metric as punctuation records starting from the sources and they will eventually reach all tasks up to the sinks". As we have seen before (section 3.3), the complete reachability of a stream process graph combined with FIFO channels guarantees that such punctuation events will reach every task in the same order that they were issued at the sources.

Low watermarks can be issued using various strategies (e.g., on periodic time intervals [23, 21]) at the source tasks of a graph, however, their underlying guarantees and progress metric derivation logic across streams is common. In Figure 6.4 we show an example of the core mechanism from the perspective of a physical task that consumes different respective input streams over time. Each data stream is enriched with periodic watermarks that indicate a progress metric which is satisfied for the rest of that particular stream. For example if a watermark has value 20, it means that no record with a timestamp $t < 20$ will be observed from that point on, on that respective stream. A task that consumes multiple streams updates its progress metric based on the most conservative value of the progress metric (i.e., the minimum of the last watermarks seen across input streams so far). Given a starting value of $t = 0$ in the example, the local progress metric is updated once the minimum watermark across input streams is higher than 0. That occurs at the time watermark with $t = 22$ is observed and the metric is updated to 20 (since $\mathtt{min}([20, 22])$). The mechanism continues in the same trivial fashion, keeping the latest watermark observed per channel and updating its local progress metric.

We add progress tracking and progress metric processing as a simple extension to our original process model presented in chapter 3. The special logic for watermark events is summarized in algorithm 9. The extension includes the following important change: in addition to the default `process` logic on input record we include the `processOnTime` which indicates to the application a (monotonic) change in a progress metric (T). Similar to the `process` function, the `processOnTime` can equally

Algorithm 9: Process Logic for Low Watermarking

```
(𝕀_p, 𝕆_p) ← configured_channels;
s_p ← ∅;                                  ▷ volatile local state
T ← 1_W;                                  ▷ the local progress metric
watermarks ← {};                          ▷ last watermarks received per-channel
foreach in ∈ 𝕀_p do
    watermarks(in) ← 1_W;
/* Common Task Logic                                                        */
Upon ⟨rcvd, W(ts)⟩ on in ∈ 𝕀_p
    tmp ← T;
    watermarks(in) ← ts;
    T ← min(watermarks.vals);
    if tmp ≠ T then
        s_p ← processOnTime(T, s_p, 𝕆_p);
        foreach out ∈ 𝕆_p do
            out → ⟨send, W(T)⟩;
Upon ⟨rcvd, m⟩ on in ∈ 𝕀_p
    . . .;
```

lead to output messages as well as volatile state change according to the operator logic. Given that watermarks arrive in-order per FIFO channel yet, unordered across channels this method ensures that a task always holds the most conservative approximation of the progress metric at hand. Initially, every watermark is set to the identity (minimum) value for the watermark 1_W and for every input value, the progress metric is updated to the corresponding minimum. Upon change in T the `processOnTime` is invoked (line 13) and a new output watermark ($W(T)$) is broadcasted to all output channels so that the rest of the processing graph updates its value (line 15).

Usage Example: We can demonstrate the usage of low watermarking via an event-time tumbling stream window with a slide of 10sec and we assume that all records carry timestamps that represent seconds, for simplicity. The initial progress metric of our window (Win) operator is currently set to 0 and computation happens out-of-order. In Figure 6.5(a) the operator receives a record with $t = 12$ and goes ahead to start a new stream window for the interval $[11 - 20]$ despite the fact that window $[1 - 10]$ is not yet complete. Computation continues in an out-of-order fashion even at the point where the operator receives a low watermark from one of its channels indicating $W = 22$ (Figure 6.5(b)). Its progress metric remains as $W = 0$ since no watermark above zero has yet been received from its other

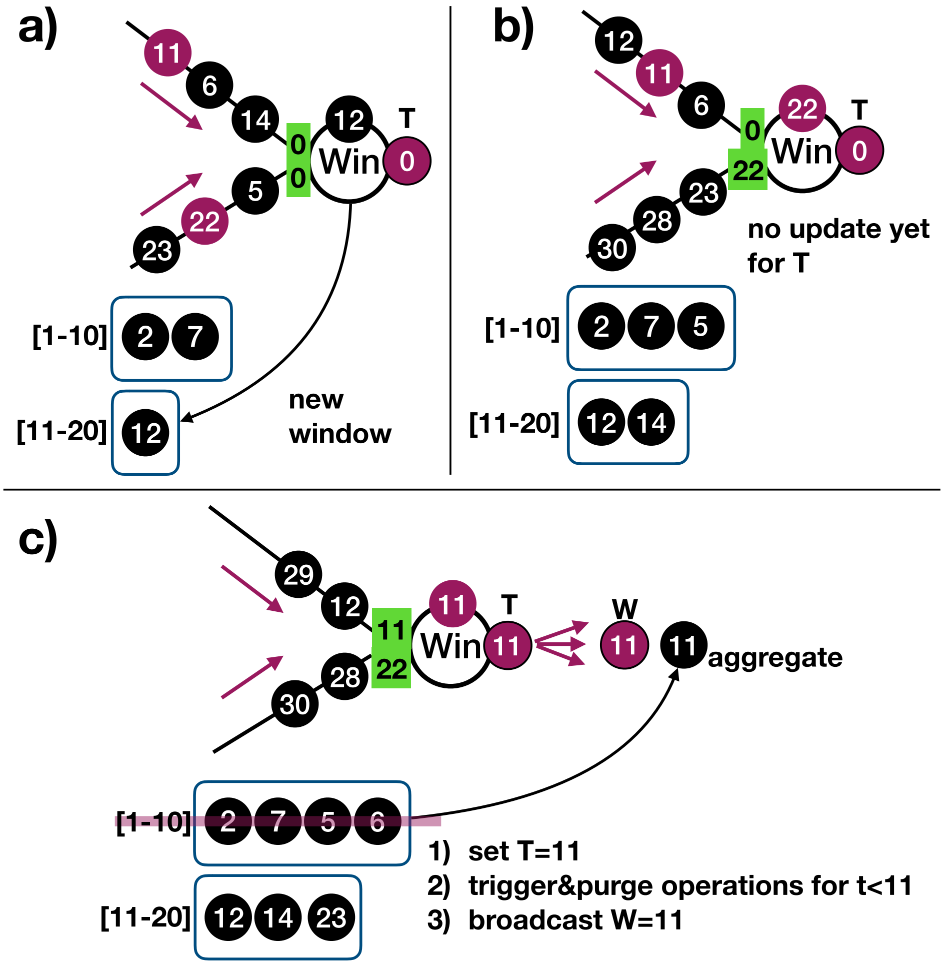


Figure 6.5: An Example of Low Watermarking on Tumbling Window (10sec).

channel. Finally, a progress metric change occurs in Figure 6.5(c) once the operator receives a watermark from the other pending channel indicating $W = 11$. Since $min([22, 11]) = 11$ its local progress metric updates to 11 and that triggers the completion of the operation of its window in range $[1 - 10]$ since $10 < 11$ given the guarantee that no more records below time 11 will arrive in the future. The operator then goes ahead and ouputs the result of the computation (tagged with the completion time $t = 11$), purges the state of the window and finally, broadcasts a watermark for $W = 11$. That aids the downstream operators to infer the progress change as explained before in algorithm 9.

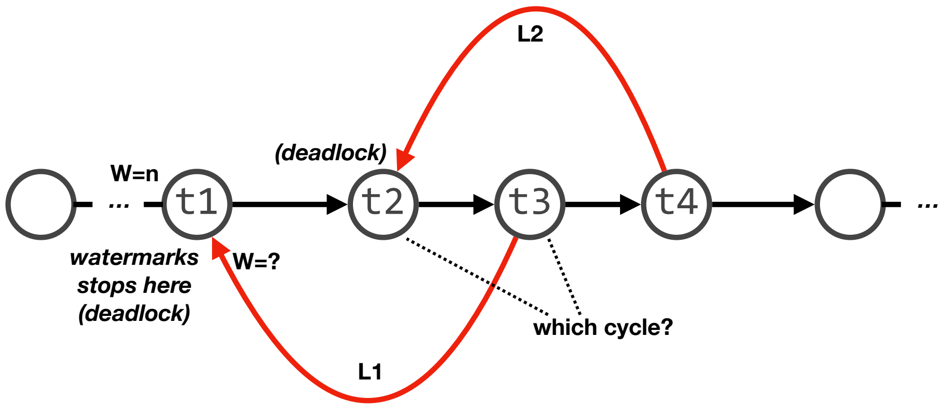


Figure 6.6: Issues of Low Watermarking with Arbitrary Cycles.

6.3.2 Iterative Processes under the lens of Out-Of-Order

Distributed iterative processes, as described earlier in this section through the `iterateBSP` programming abstraction, can be viewed as a special progress-driven blocking operation in the context of data streaming. Similarly to a progress-based window aggregation, an iterative process consists of the step function L, an operation that can be computed in-parallel (across different instances of L) and out-of-order (e.g., when multiple iterative processes are initiated in parallel). In addition, we described the Bulk and State-Synchronous parallel synchronization primitive (`sync`) that is based on a global progress metric, the distributed progress of the iterative process measured in complete supersteps. In principle, total order within iterative supersteps can be replaced by a progress metric that eventually guarantees that a stable property is satisfied, in this case the parallel completion of a superstep. Finally, similarly to a window aggregation, each iterative process is guaranteed to complete and terminate, purge its state and yield an output result. With the above intuition we go further and identify current stream processing model restrictions that need to be resolved in order to allow full support for arbitrary iterative progress metrics embedded within a stream process graph.

6.3.2.1 Structural Limitations and Problem Intuition

Given that applications in stream processing are stand-alone deployable graphs of tasks, the ability to use physical cycles in the graph is paramount. However, there are unresolved implications if we try to combine OOP and cyclic graphs. In fact, all existing low watermarking adaptations [22, 9, 55, 46] make a critical assumption, that the computational graph is acyclic. There are several reasons why this has been the case. If we consider arbitrary cycles in a stream process graph (even after the transformations considered in chapter 3 Section 3.4.4.4), the feasibility of the watermarking protocol as well as the stable property that progress metrics represent can be violated. We refer to some of the core issues in the dataflow graph example depicted in Figure 6.6. In this example, we have two arbitrary cycles L1 and L2. In both cyclic cases, if we consider the low watermarking protocol from algorithm 9 as it is we will have a clear deadlock situation. Since the out-of-order logic allows computation to be invoked arbitrarily records will be processed continuously, however, blocking operations would never complete past task `t1` in the graph. That is due to the fact that `t1` will wait until a watermark higher than 1_W arrives from the feedback channel, though this will never occur since that condition itself depends on `t1` to disseminate its low watermark. In addition to the deadlock problem, having old circulating records alongside expired watermarks would repeatedly violate the stable property of progress metrics. Finally, with an unstructured control flow such as the case depicted in Figure 6.6 where `t2` and `t3` belong to two cyclic components L1 and L2 different events that go around that graph cannot have computational context (e.g., which cycle and which superstep of an iterative process they belong to). The underlying reason is the lack of program structure and scoping (similarly to unstructured programming using `goto` statements). These observations motivate a few fundamental changes to our original stream process model: 1) There is a need for structured stream graphs that have clear scope and can support nesting, and 2) Watermarks need to be extended with nested progress metrics to eliminate the possibility of deadlocks and properly reflect the progress of a computation in respective parts of the process graph.

6.3.3 Model Extensions for Cyclic Progress

We present a set of stream process model additions as well as restrictions that can facilitate the implementation of various kinds of iterative processing operators. Our approach integrates well with the out-of-order processing paradigm and requires no central coordination. In fact, the default low watermarking scheme remains as is, while our additions focus solely on structural properties and progress tracking within cyclic subcomponents of a stream application.

6.3.3.1 Scopes, Structured Loops and Progress Timestamps

We seek to incorporate cyclic progress metrics in addition to the default application time metric in order to allow operators to track and drive iterative computation. The Progress Timestamps, introduced in Section 6.2.1.2 can be used to carry arbitrary nested progress measures as well as for determining a termination criterion. However, we first need a graph structure that can facilitate them, namely *scopes*. Scoping is essential for providing selective access to progress metrics within respective graph regions, similar to the usage of lexical scopes in programming

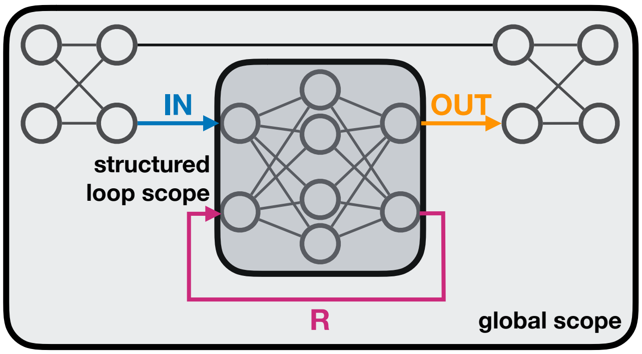


Figure 6.7: A structured loop within a stream process graph

languages for restricting access to local variables. By default, every task belongs to the global scope where progress metric is observed (i.e., application-time). Iterative computation and hence progress for iterative computation is inherently relevant to the subgraph that executes the particular logic of the iteration. To allow structured subgraphs, we disallow arbitrary feedback channels (e.g., such as the ones showed in Figure 6.6) and we restrict nested subgraphs through the use of a `structured loop` operator. In this section, we will examine the basic underlying properties of structured loops in an implementation-agnostic way, while in section 6.4 we will show how they can be used to provide programming support for iterative processes on stream windows using Apache Flink.

Structured Loops: Logical operations on a process graph can be declared as transformations on streams. For example, a map operator can be defined as such:

`map:(stream[IN],` mapFun : (`IN` $\longrightarrow$ `OUT`)) $\longrightarrow$ `stream[OUT]`

The structured loop is a special type of a stream operator that defines a new scope with a built-in feedback stream for composing iterative logic and a termination condition, defined as such:

`loop:(stream[IN],` loopFun,termFun) $\longrightarrow$ `stream[OUT]`

loopFun:(`stream[IN],stream[R]`) $\longrightarrow$ (`stream[R], stream[OUT]`)

termFun: $(R, P) \longrightarrow$ bool

In contrast to known unary and binary stream operators, the loop operator effectively takes as an argument a stream graph (the Loop DAG) that adheres to a strict specification: its operators can consume one regular input stream (`stream[IN]`), produce a regular output stream (`stream[OUT]`), while also allowing to produce and consume a feedback stream (`stream[R]`). The termination function (termFun) encapsulates the condition for completing an iterative process as we will cover in more in Section 6.3.4.5. In Figure 6.7, we depict a structured loop and its scope,

Operator	Progress Operation	Description
L^{IN}	*nest*: $\mathbb{N}^n \to \mathbb{N}^{n+1}$	Initializes scope's progress metric
L^{OUT}	*unnest*: $\mathbb{N}^{n+1} \to \mathbb{N}^n$	Triggers final (unnested) output
L^T	*incr*: $\mathbb{N}^{n+1} \to \mathbb{N}^{n+1}$	Increments scope's progress metric
L^H	*term*: $\mathbb{N}^{n+1} \to \mathbb{N}^{n+1}$	Forwards or terminates a progress metric

Table 6.2: Logic of Special Operators for Progress Timestamps

as part of a stream process graph. From a top down view the structured loop can be seen as a regular operator that consumes input records and watermarks and respects regular watermark-based execution. Nesting is allowed, yet it is hidden from the outer scope. That is, given a user-defined loop DAG, any operator in the DAG can itself be a structured loop.

6.3.3.2 Overview of Embeddings for Progress Timestamps

At the global scope, progress is captured as a single metric, similar to a Progress Timestamp (see Section 6.2.1.2) having a single value. However, each structured scope introduces its own local progress metric. With this design, we seek for the right embeddings in a stream process graph that can automatically convert progress metrics (i.e., nesting, unnesting and incrementing) on record timestamps and watermarks that transit across scope boundaries. Operators in a scope should be granted access to a local metric and the same should apply for all nested scopes in the graph. Structured loops provide a clear foundation to incorporate the right progress operations on scopes, having an explicit entry (`stream[IN]`) and exit (`stream[OUT]`) from a particular scope, as well as a predefined feedback (`stream[R]`). As depicted in the example of Figure 6.8, all necessary transformations on progress timestamps can be implicitly supported through a set of special operators created per defined scope. Since this model builds on our original work of modeling back-edges (chapter 3 Section 3.4.4.4) we assume the existence of an iteration head L^H and tail L^T operator that support the back-edge record forwarding logic within each respective scope of a structure loop L. We further introduce two additional operators per loop scope, namely the entry L^{IN} and exit L^{OUT}. Each of these operators transforms message timestamps across the boundaries of a scope as summarized in Table 6.2.

The set of implicit progress operations makes sure that all operators residing within a certain scope will access the same progress metric and that no irrelevant operators outside that scope can access the loop's internal progress metric. All operations within the scope are being executed in an out-of-order fashion, while tracking partial order metrics (explained in subsection 6.3.4). Furthermore, the `incr` operation enforces the advancement of all records and progress metrics (i.e., watermarks) that transit back to the beginning of the structure loop's subgraph in order to separate computation between steps and guarantee correctness for the

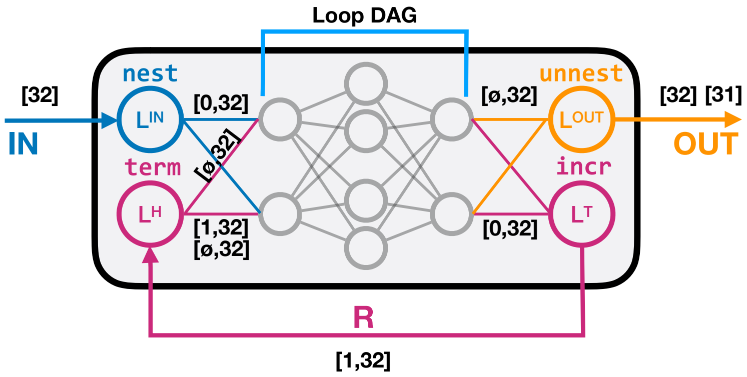


Figure 6.8: Anatomy of a structured loop and its embeddings

progress tracking algorithm (Section 6.3.4.2). Finally, the loop head L^H operator executes the termination condition and issues a special termination watermark $(ø)$ when an iterative operation has been completed in order to complete pending computation, emit results and garbage collect state (Section 6.3.4.5). In Section 6.3.4.3 we summarize all the special logic of every embedded operators in a structured loop.

Example: In the example depicted in Figure 6.8, a message (record or watermark) with a progress timestamp $P = [32]$ arrives into the scope (from the global scope). The L^{IN} operator converts P to $[\mathbf{0}, 32]$ adding a new local progress metric initialized at value 0. When a message with the same timestamp traverses through the feedback stream the loop tail operator L^T will increment the local progress metric, yielding timestamp $P = [\mathbf{1}, 32]$. We then observe two possible cases: if the termination condition invoked by operator L^H is not satisfied then the message timestamp is forwarded as is. However, if the termination condition yields "true" then L^H disseminates a termination watermark $[\mathbf{ø}, 32]$ that indicates the completion of a computation. Finally, when the L^{OUT} operator receives the termination watermark with time $[ø, 32]$ it releases all (stored) pending results at the output stream of the structured loop with unnested timestamps.

In the rest of this section, we are going to expand on the previous general description of structured loops and address all the internal mechanisms and actual guarantees in further detail.

6.3.4 Decentralized Progress Tracking for Partial Event Order

We will now examine how all the primitives we have introduced so far can facilitate correct progress tracking of a single progress metric within a structured loop. Our general approach focuses on partial iterative execution order, a concept that we motivate below.

Motivation: In Section 6.3.1 we defined progress metrics as a sufficient means of capturing the progress of a *totally ordered* attribute such as event time in a distributed stream execution. With each nested structured loop, we effectively introduce a new progress metric capturing a global *partial* order of messages for each distinct timestamp that enters its scope. This yields the prospect of achieving out-of-order processing across distinct timestamps as well as across parallel supersteps executed per timestamp. We argue that partial order is sufficient for progress tracking in the context of implementing iterative processes on data streams. That is, first, due to the fact that bulk iterations need a fixed data set to operate on (i.e. the entry set X) instead of an unbounded stream of changes. Evidently, it suffices to scope iterations such the restricting their computation to the level of each stream window (or "epoch", as defined in the Naiad [20] project) and since windows can be processed out-of-order, a partial order tracking metric is sufficient for that purpose. Finally, another reason we do not consider total order is the unbounded nature of iterative processes. Implementing iterative processes is the primary goal of this work, yet each iterative process can take an arbitrary number of steps to finish. If we block execution until every consecutive iterative process is complete we effectively turn our asynchronous execution into a sequential blocking execution which does not conform with the data streaming and out-of-order processing paradigms in general.

6.3.4.1 Partial Progress Timestamp Order

We distinguish two parts in every progress timestamp: 1) its current progress $P^T = P.progress$ and 2) its context $P^{ctx} = P.unnest$:

$$P = [\overbrace{p^n}^{P^T}, \underbrace{p^{n-1}, p^{n-2}, \dots, p^0}_{P^{ctx}}]$$

We are generally interested to track the progress of messages in respect to a specific context. This type of partial order relation is described in Definition 6.3.2.

Definition 6.3.2. Partial Timestamp Order: In-transit messages within the same scope are partially ordered over a progress timestamp $P \in \mathbb{N}^n$ as follows: For $P, P' \in \mathbb{N}^n : P \geq P'$ iff $(P^{ctx} = P'^{ctx}) \wedge (P^T \geq P'^T)$

We further consider the termination time (ø) as the maximum time. That is, there exists no $v \in \mathbb{N}$ such that $v \geq ø$. Now, given a *partial order* as defined above, we can reduce the scope of total-order progress metrics to a partial order metric within structured loops which is summarized in Definition 6.3.3.

Algorithm 10: Context-Based Low Watermarking

```
$(\mathbb{I}_p, \mathbb{O}_p) \leftarrow$ configured_channels;
$s_p \leftarrow \varnothing$;                                        ▷ volatile local state
$T \leftarrow \{\}$;                          ▷ the local progress metric per context
$watermarks \leftarrow dict()$;      ▷ latest metrics received per context and channel
/* Common Task Logic                                                  */
Upon $\langle rcvd, W(P) \rangle$ on $in \in \mathbb{I}_p$
    if $T(P^{ctx}) = nil$ then
        $initialize(P^{ctx})$;
    $tmp \leftarrow T(P^{ctx})$;
    $watermarks(P^{ctx})(in) \leftarrow P^T$;
    $T(P^{ctx}) \leftarrow min(watermarks(P^{ctx}).vals)$;
    if $tmp \neq T(P^{ctx})$ then
        $(s_p, c_{pq}) \leftarrow processOnTime(T(P^{ctx}), P^{ctx})$;
        foreach $out \in \mathbb{O}_p$ do
            $out \rightarrow \langle send, W(T(P^{ctx}) :: P^{ctx}) \rangle$;
    if $T(P^{ctx}) = ø$ then
        $watermarks(P^{ctx}) \leftarrow T(P^{ctx}) \leftarrow nil$;      ▷ progress purging
Fun initialize(ctx)
    $watermarks(ctx) \leftarrow \{\}$;
    foreach $in \in \mathbb{I}_p$ do
        $watermarks(ctx)(in) \leftarrow 1_W$;
    $T(ctx) \leftarrow 1_W$;
```

Definition 6.3.3. Partial Order Progress Metric: Given a partial order of progress timestamps within the scope of a structured loop L, a *progress metric* $W^{ctx} : T \rightarrow \mathbb{N}$ of T is a stable property that indicates $v \in \mathbb{N}$ if no record with $P^T \leq v$ will be processed for context P^{ctx}.

6.3.4.2 Context-Based Low Watermarking

Low watermarking can be used for partial-order progress tracking simply by running a new instance of the protocol per unique context observed. That means that each operator would need to maintain a progress metric per context as well as issuing watermarks per progress update that occurs on a specific context. Furthermore, when the computation on a specific context has finished (indicated by a progress termination symbol ø) the operator would have to purge the respective progress metric. We first discuss the updated logic for low watermarking in nested scopes

Algorithm 11: Operator Logic for L^{IN}

```
1: Upon $\langle rcvd, msg : W(P)|record(P)\rangle$ on $in \in \mathbb{I}_p$
2:     $out \rightarrow \langle send, msg(P.nest)\rangle$;        ▷ received directly (no shuffling)
```

Algorithm 12: Operator Logic for L^{H}

```
1: /* Logic for $L^H$                                      */
2: Upon $\langle rcvd, record(P)\rangle$ on $in \in \mathbb{I}_p$
3:     $termFun(record, P)$;                                ▷ indicate record
4:     $out \rightarrow \langle send, record(P)\rangle$;
5: Upon $\langle rcvd, W(P)]\rangle$ on $in \in \mathbb{I}_p$
6:     if $termFun(nil, P)$                ▷ indicate watermark for termination
        then
7:         foreach $out \in \mathbb{O}_p$ do
8:             $out \rightarrow \langle send, W(\emptyset :: P^{ctx})\rangle$;
```

and then prove how the graph embeddings we introduced earlier provide the necessary implicit progress-related guarantees.

Core Algorithm: In algorithm 10 we summarize the updated logic executed by every task in the system to track progress. To allow partial order execution in nested scopes, we extend all basic operations on time to be scoped by context. Starting with local progress metrics, each task will now need to keep a separate progress metric initialized per context (line 18) and issue watermarks (line 15) that contain complete progress timestamps $W(P)$, where P encapsulates the context (P^{ctx}) and the progress metric for that context (P^{T}). Furthermore, we extend `processOnTime` to include the context of each progress update so that operators can scope computation per particular context (e.g., to partition and purge state per context). Furthermore, when the local progress metric of a specific context reaches the termination value (indicated as ø), all state is being purged (after computation is triggered through the `processOnTime` invocation). Finally, in the common case of tasks that are not part of a structured loop (in the general scope) the logic is identical to the original algorithm since only one context is defacto considered, that of an empty vector [].

6.3.4.3 Implicit Logic on Embedded Operators

In addition to the general algorithm 10 we also specify the logic on each of the embedded operators of a structured loop, namely L^{IN}, L^{OUT}, L^{H} and L^{T}.

$Entry[L^{IN}]$ **(algorithm 11):** The entry operator is completely stateless and receives

Algorithm 13: Operator Logic for L^T

```
1: Upon $\langle rcvd, record(P)\rangle$ on $in \in \mathbb{I}_p$
2:     out $\rightarrow \langle send, record(P.incr)\rangle$;
3: Upon $\langle rcvd, W(P)\rangle$ on $in \in \mathbb{I}_p$
4:     if $T(P^{ctx}) = nil$ then
5:         initialize($P^{ctx}$);
6:     $tmp \leftarrow T(P^{ctx})$;
7:     $watermarks(P^{ctx})(in) \leftarrow P^T$;
8:     $T(P^{ctx}) \leftarrow min(watermarks(P^{ctx}).vals)$;
9:     if $tmp \neq T(P^{ctx})$ then
10:        out $\rightarrow \langle send, W(T(P^{ctx}).incr)\rangle$;        ▷ forwarded directly to $L^H$
11:    if $T(P^{ctx}) = \emptyset$ then
12:        $watermarks(P^{ctx}) \leftarrow T(P^{ctx}) \leftarrow nil$;
```

directly all records and watermarks from the outer scope. Given that no data shuffles are required to create a scope, we assume that this operator is directly connected to a data-parallel operator such as a time window and receives records and watermarks derived in a single stream partition (similarly to a window aggregation). All the entry operator does is therefore to *nest* the progress timestamps that are then forwarded into the structured loop.

Head[L^H] **(algorithm 12):** Each parallel instance of the head operator is directly connected to a respective parallel tail operator instance. Since there is a single input stream, no progress metric needs to be derived at this point since all received watermarks are already derived from the tail operator. However, the loop head operator is crucial for terminating an iterative progress tracking protocol by consulting the user-provided termination function (`termFun`) which inspects all forwarded messages, watermarks and records in the respective partition. This is also where the logic of a termination condition (as discussed in section 6.2) can be employed upon a watermark. Mind that if a subset of the instances of L^H operators indicates termination on a context, it is not enough to terminate it completely. As the default low watermarking protocol goes, the progress metrics of the internal operators will only update to the terminal value ($\emptyset$) if *all* of the stream partitions have satisfied the termination condition.

Tail[L^T] **(algorithm 13):** The tail emulates one endpoint of a feedback channel and intercepts all records and watermarks which are sent to the next step of an iteration. For simple records, the tail operator simply increments their current progress timestamp and forwards them to the next iteration step via its corresponding head operator. When it comes to watermarks the tail operator first derives that

Algorithm 14: Operator Logic for L^{OUT}

```
1: Upon $\langle rcvd, record(P)\rangle$ on $in \in \mathbb{I}_p$
2:     $s_p \leftarrow process(record, P^{ctx}, \mathbb{O}_p)$;        ▷ keep until purged
3: Upon $\langle rcvd, W(P)\rangle$ on $in \in \mathbb{I}_p$
4:     if $T(P^{ctx}) = nil$ then
5:         $initialize(P^{ctx})$;
6:     $watermarks(P^{ctx})(in) \leftarrow P^T$;
7:     $T(P^{ctx}) \leftarrow min(watermarks(P^{ctx}).vals)$;
8:     if $T(P^{ctx}) = \varnothing$ then
9:         $(s_p, c_{pq}) \leftarrow triggerAndPurge(P^{ctx})$;        ▷ emit data & watermarks
10:        $watermarks(P^{ctx}) \leftarrow T(P^{ctx}) \leftarrow nil$;
```

progress has been made (i.e., all wateramarks of a step have been received) and then disseminates further the incremented progress metric for the next step.

$Exit[L^{OUT}]$ **(algorithm 14):** The exit operator collects outgoing records and re-sequentializes output and watermarks according to the progress metric of the outer scope. Without any loss of generality, it can be seen as a special window operator that continuously processes (stores) output records per context (line 2) and finally triggers them and purges that state once no preceding context computation is pending (line 9), ordered by the outer scope metric. Note that the exit operator does not forward watermarks outside the scope. That is because a context is only considered terminated once all of the iterative computational steps have finished for the that context and every preceding context based on the outer progress metric. The `triggerAndPurge` function simply appends a watermark after each completed output, similarly to the logic of window triggers analyzed before in Figure 6.5.

6.3.4.4 Approach Correctness

The context-based watermarking scheme together with the embedded graph operations can guarantee that the partial order metric that is tracked in each scope will not be violated. That is, for every progress value $v \in \mathbb{R}$ indicated on a context $a \in \mathbb{N}^{n-1}$ it should guarantee that no record with progress timestamp P where $P^T \leq v$ and $P^{ctx} = a$ will be processed in the rest of the execution.

Since correctness is bound to partial order let us consider only the events that reside in context a. Within the subset of an execution that is relevant to context a the protocol has an identical logic to the original low watermarking algorithm. i.e., "the lowest, most conservative value across the latest watermarks received across input channels (streams) decides the actual value of the progress metric". The only difference is that it executes a parallel instance per context. Furthermore, the core

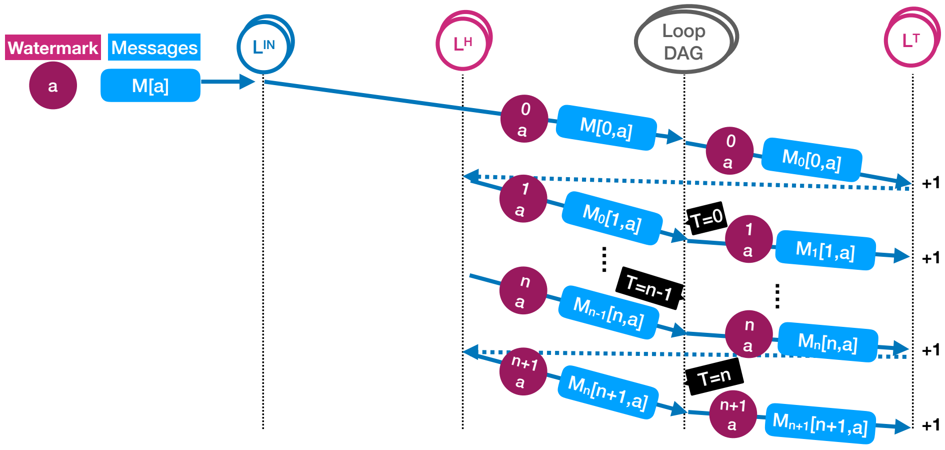


Figure 6.9: Messages and Watermarks per-context in a Structured Loop

low watermarking strategy has proven to guarantee correctness in a distributed acyclic graph [22] with FIFO channels since *a watermark never precedes a record that has a lower timestamp*. What we need to prove here is that the special embedded logic within a structured loop can still maintain that invariant.

Let us consider the full set of messages M with $P = a$ that enter a structured loop M from different streams. Given the default low watermarking logic these messages will always precede a watermark $W(a)$. If we now break down the distributed execution within a structured loop, events follow a strict flow of steps, as depicted in Figure 6.9. We will discuss how our invariant is not violated via induction on the transitive closure of messages produced in the iterative execution.

Step 0: Every message passes through the order-preserving entry operator (L^{IN}) that turns a into a context and adds a progress metric of value 0 on every message that is being forwarded, followed by a watermark marking the completion of the input with a nested metric value of 0. The computation in the first iterative step occurs within the user-provided graph which, itself can be considered a DAG (nested structured loops also behave as a single operator). Consequently, given the original protocol logic, the transitive closure of the output generated by the loop DAG is nothing more than a set of messages M_0 at time 0 followed by forwarded watermarks of time 0 which end up in the loop tail. Our invariant is so far guaranteed until every message of step 0 arrives at the tail.

Step 1: The loop tail increments message timestamps and forwards them back to the DAG through the head operator in strict FIFO order. That means that effectively, the transitive closure of messages entering the second iteration step would be M_0 and progress 1 followed by watermarks of time 1. Mind that order within M_0 is

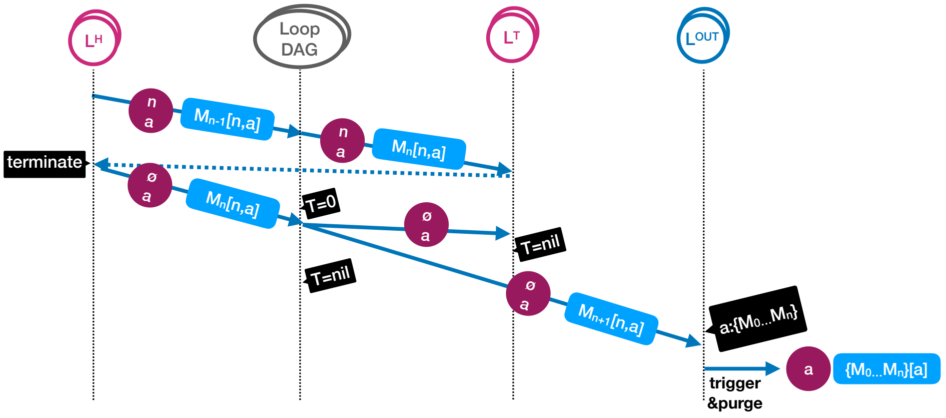


Figure 6.10: Context Termination in a Structured Loop

not important (and typically not maintained through channel shuffles in the DAG), as long as watermarks do not indicate incomplete steps (e.g., at this point if W(t>1 :: a)). The process continues as in step 0, maintaining the invariant.

Step n: Similarly, if we consider M_n, the complete output set of messages that a loop DAG generates at step n, it naturally precedes watermarks of time n which in turn are fed into the DAG again with incremented timestamps.

Via induction on the transitive closure of messages produced during a distributed execution and the induction steps discussed above, it is ensured that a watermark $W(n :: a)$ would always follow the transitive closure of messages $\{M_0, \ldots, M_n\}$. Since local progress metrics cannot surpass input watermarks our main invariant is always guaranteed internally in a structured loop, per context.

6.3.4.5 Termination and Garbage Collection

Our structured loop scheme needs to guarantee termination, based on a user-defined criterion (using termFun). Termination occurs when all tasks within a structured loop are notified regarding the completion of the computation for a specified context a. In our model, we used a special progress metric (ø) to disseminate the notion of completion. Upon notification of completion the embedded as well as the special operators that keep progress (L^T, L^{OUT}) can trigger their pending computation as well as purge their stored progress metrics and input watermarks for the completed context a. Assuming that the termination criterion is guaranteed to be eventually satisfied on all loop head instances, we start from there to examine how completion notifications are disseminated, also depicted in Figure 6.10.

According to the logic executed by L^H the termination criterion is determined by the loop head operators at the reception of a watermark (the loopFun is invoked per

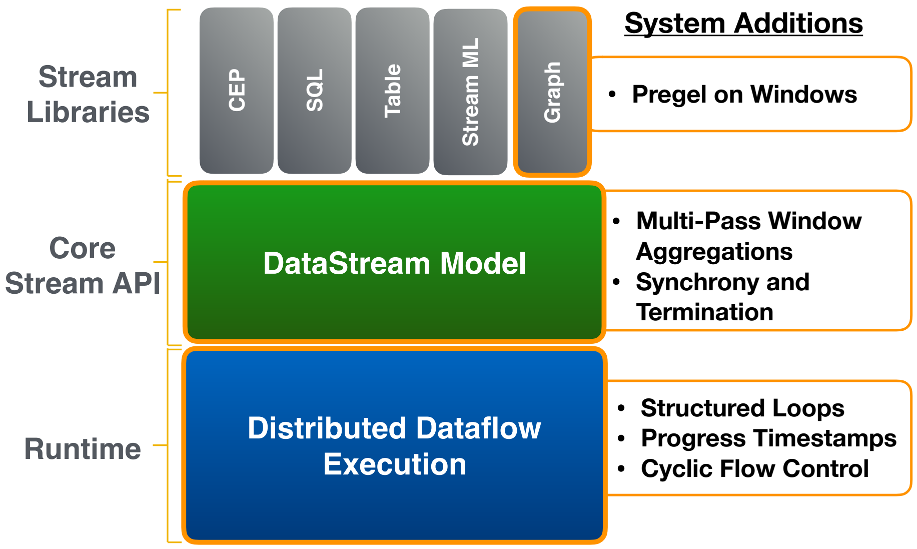


Figure 6.11: Overview of System Additions (based on Apache Flink 1.5)

record to allow for data-dependent termination criteria build from the data). Let $n+1$ be the metric that invokes a termination process. Instead of $W(n+1 :: a)$, L^{H} would instead disseminate to its output channels a termination metric $W(\varnothing :: a)$. The dissemination of termination metrics is identical to the one of regular watermarks. This means that termination watermarks would be broadcasted throughout the DAG, eventually reaching every task and succeed the last messages outputted from the purged operations. Eventually, all termination watermarks would reach all tasks of the exit operator L^{OUT} to trigger and purge all pending output from that context (e.g., $\{M_0, \ldots, M_n\}$). Same applies for the tail operator L_T which needs to garbage collect its progress metrics for context a.

6.4 Iterative Processes over Windows on Apache Flink

In this section, we present programming model that provides support for iterative processes on stream windows, implemented as an extension of the Apache Flink framework. From the user-facing programming framework, iterative processes can be seen as a special case of window aggregation (chapter 5) that requires multiple passes of the data.

Our design extends several layers of the Apache Flink stack, as depicted in Figure 6.11. At the runtime level, we provided native system support for structured loops and progress timestamps, according to the exact specification covered in

Iteration Primitive	**Definition**	**Description**
Termination	Termination.Fixpoint Termination.Fixed($n \in \mathbb{N}$)	Loop Termination Criterion
Synchronization	Synchrony.Stale($n \in \mathbb{N}$) Synchrony.Strict	Loop Synchronization Type
Loop Key	$R \to K$	Key Extractor for Structured Loop
Entry Function	$(ctx, [IN]) \to (ctx, [R])$	Initialization Logic
Step Function	$(ctx, [R]) \to (ctx, [R])$	Iteration Step Logic
Finalize Function	$(ctx) \to (ctx, [OUT])$	Finalization Logic after Termination

Table 6.3: Configuration and Computation Primitives for Window Iterations

subsection 6.3.3. Furthermore, we extended Flink's core flow control logic to eliminate the possibility of deadlocks within cycles. As a proof of concept, on top of structured loops we implemented an iterative operator for stream windows. Our design encapsulates all basic primitives presented in section 6.2, namely different termination conditions (c), synchronization ($sync$) and user-defined step functions (L) built on top of structured loops and Flink's managed state (see chapter 4). Finally, in addition to Flink's domain specific stream libraries (i.e., complex event processing and Tables-Stream SQL) we composed a graph stream domain-specific library that makes use of window iterations to support a vertex-centric graph API similar to Pregel's [17].

6.4.1 Programming Model

We will first introduce the programming primitives of window iterations, followed by concrete usage examples and higher-level models that can be built on top such as Pregel's vertex-centric graph processing model.

6.4.1.1 The Window Iterate Operator

The window iteration model encapsulates all primitives of an iterative process introduced earlier and is applicable to Flink's parallel stream windows. Each iterative process can be described by a termination criterion, synchrony option, a key that partitions the data and a set of three core aggregation functions: *entry, step* and *finalize* as summarized below:

```
val stream: DataStream[IN] = ...
stream.keyBy(...).window(...)
    .iterate(<Termination>,<Synchrony>,<Key>,<Entry>,<Step>,<Finalize>)
```

In Table 6.3 we summarize the description of these primitives, including the system-provided configuration properties for termination and synchronization.

LoopContext Properties	Definition
key: K	The current key of the computation
localProgress: $\mathbb{N}$	The current superstep of the local computation
globalProgress: $\mathbb{N}$	The last parallel complete superstep
eventTime: $\mathbb{N}$	The event time (ctx) of local computation
loopState: ManagedState	keyed state, purgeable per iterative process
persistentState: ManagedState	keyed state, persistent across iterative processes

Table 6.4: Variables Accessible under the loop context (ctx)

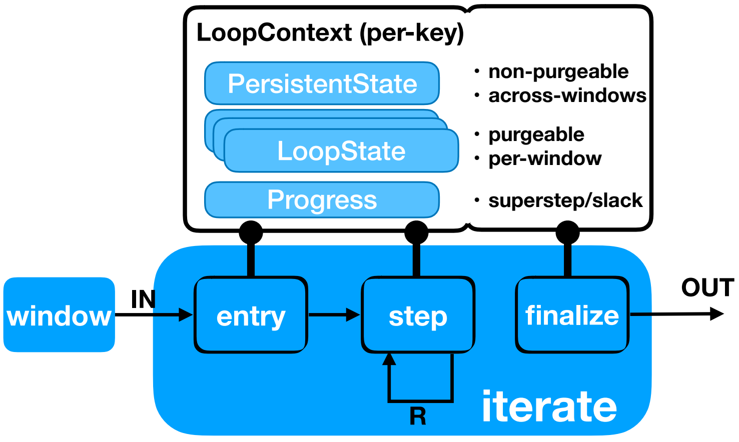


Figure 6.12: Window Iterate Operator Internals

Similarly to Flink's existing API[3], access to managed state and metadata of an aggregation are provided through a special LoopContext object (ctx in Table 6.3), that grants access to the key of the computation, progress metrics and two types of read-and-write managed state: 1) *loopState* for defining variables that become purged at the end of an iterative process of a window and 2) *persistent state* for variables that are shared across windows. Table 6.4 provides a summary of all these accessible properties. It should be noted that since both types of state are managed, the system's snapshotting (chapter 3) and epoch commit protocol (chapter 4) ensure its consistent processing.

In Figure 6.12 we depict the logical flow between the aggregation functions within the window iterate operator and further describe their usage below.

Entry Function: The Entry function is called once (per-key) when the input window triggers and allows for initializing the state of an iterative process (e.g., defining

[3]https://ci.apache.org/projects/flink/flink-docs-release-1.5

initial state and storing it in the loop context). Furthermore, the whole iterative computation gets kickstarted within the entry function by emitting the messages [R] that will be fed back to the first superstep.

Step Function: The step function receives all feedback updates generated per-key in the previous superstep and generates new updates for the next superstep. Furthermore, it has access to the LoopContext of the window iteration, being able to read and write variables within both purgeable and persistent managed state.

Finalize Function: The finalize function is invoked when the termination criterion is satisfied (e.g., when no changes are submitted in the last step during a fixpoint aggregation). In that case, The final output of the aggregation has to be derived from the context, either by reading the the latest window state before it gets cleared, or from the persistent state. This also gives a convenient way to incorporate any further necessary changes to the persistent state consistently at the completion of a window iteration.

Listing 6.3: Connected Components: Example of FixPoint Iteration

```scala
case class GraphEdge(from:Long, to:Long)
case class VComponent(id:Long, component:Long)

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)) //add inverse
  .keyBy(vertex => vertex.id) //Scope Computation by Key
  .timeWindow(1 Min)
  .iterate(Termination.Fixpoint, Synchrony.Strict,
    (vComponent => vComponent.id), //Loop Key
    (ctx:LoopContext,in:Iterable[GraphEdge],out:Collector[VComponent]) =>
       {  //ENTRY FUNCTION
          ctx.loopState("neighbors").setList(in.map(e => e.to));
          ctx.loopState("cc").setValue(ctx.key)
          ctx.loopState("neighbors").foreach(n => out.collect(VComponent(n,ctx.key)))
       },
    (ctx:LoopContext,in:Iterable[VComponent],out:Collector[VComponent]) =>
       {  //STEP FUNCTION
          val newcc = in.minBy(c => c.component).component
          var cc = ctx.loopState("cc")
          if(cc.value != newcc){
             cc.setValue(newcc)
             ctx.loopState("neighbors").foreach(n => out.collect(VComponent(n,newcc)))
          }
       },
    (ctx:LoopContext,out:Collector[VComponent]) =>
       {  //FINALIZE FUNCTION
          out.collect(VComponent(ctx.key, ctx.loopState("cc").value))
       });
```

6.4.1.2 Usage Examples

We demonstrate the usage of the window iterate operator through a series of examples that make use of its basic primitives. Then, we will show examples of high-level models that are composed for domain-specific continuous applications.

Fixpoint-Iteration Example: In Listing 6.3 we show an adaptation of the iterative BSP algorithm for ConnectedComponents presented previously in Listing 6.2. Graph data is ingested in the form a an unbounded stream of edge (`GraphEdge`) objects each representing a new directional graph edge from source to target vertex id. For each edge we add its inverse edge and partition that stream and its corresponding states by vertex id. Then, we define a tumbling window that contains all edges generated within one minute of application time (inferred from external timestamps). The `iterate` function initiates an iterative process per triggered window configured with a termination function and synchronization type. In this case, we use the built-in `Termination.Fixpoint` function which terminates when no more messages are exchanged and is identical to the logic described previously in Listing 6.2. The `Strict` synchrony type corresponds to no asynchrony, executing steps in a bulk synchronous iterative fashion.

Listing 6.4: PageRank: Example of Fixed Iteration with Persistent State

```scala
case class GraphEdge(from:Long, to:Long)
case class VRank(id:Long, rank:Double)

def computeRank(w:Iterable[Double]) = 0.15/w.size + 0.85 * w.sum

val input: DataStream[GraphEdge] = getEdgeStream()
input
  .flatMap(e => List(e,GraphEdge(e.to,e.from)))
  .keyBy(edge => edge.from)
  .timeWindow(30 Sec)
  .iterate(Termination.Fixed(40), Synchrony.Strict,
    (vRank => vRank.id),
    (ctx:LoopContext,input:Iterable[GraphEdge],out:Collector[VRank]) =>
        {   //ENTRY FUNCTION
            ctx.loopState("neighbors").setList(input.map(e => e.to))
            //Start from stored rank
            val rank = ctx.loopState("rank")
            rank.setValue(ctx.persistentState("rank").value)
            ctx.loopState("neighbors").foreach(n => out.collect(VRank(n, rank.value)))
        },
    (ctx:LoopContext,input:Iterable[VRank],out:Collector[VRank]) =>
        {   //STEP FUNCTION
            val newRank = computeRank(input.map(c => c.rank))
            ctx.loopState("rank").setValue(newRank)
            ctx.loopState("neighbors").foreach(n => out.collect(VRank(n, newRank)))
        },
    (ctx:LoopContext,out:Collector[VRank]) =>
        {   //FINALIZE FUNCTION
            //Log new rank
            val finalRank = ctx.loopState("rank").value
            ctx.persistentState("rank").setValue(finalRank)
            out.collect(VRank(ctx.key, finalRank))
        });
```

The main algorithm logic is encapsulated within the three iteration functions each of which is scoped by vertex id (the key used in this application). The entry function initializes the vertex state with the list of its neighbors and an initial value (line 14) while also broadcasting its value to its neighbors (line 15). These messages are then shuffled to the respective vertex partitions and processed within the first step function, at the first iteration step. Notice that the logic within the step function is identical to the original definition of the algorithm in our iterate BSP model Listing 6.2. The only implementation-specific concerns here are the usage of the loop state from the provided context object and the Collector primitive of Flink, used to flush intermediate output results incrementally. The finalize function is eventually invoked when no more messages are exchanged within a round and the final output is in fact the latest value of the vertex. Mind that the loop state is purged after the completion of an aggregation in a respective window (context), thus, if persistent application state is needed, e.g., to store and use the connected component per vertex the program should make use of the corresponding `persistentState` provided via the LoopContext.

Fixed-Iteration Example with Persistent State: In Listing 6.4 we cover another example of a graph algorithm, in this case, the iterative PageRank. The implementation is based on the original pseudocode from Listing 6.1 so we will omit its description. Instead, we focus on the distinct characteristics of this implementation, namely the usage of persistent state and fixed termination. As seen in the entry function (line 17), in this version of the algorithm a rank of an iterative process gets initialized based on the value of a previously computed rank that is fetched from the persistent state of that vertex. Similarly, within the finalize function the persistent value for the rank of that vertex is updated to the newest finally computed rank (line 29). When it comes to termination, a maximum number of steps is set to stop the iterative process solely based on its progress metric.

6.4.2 High-level APIs

The window iterate operate operator can be used as a building block for implementing already known as well as new iterative programming models on data stream processing. As is, a window iteration requires the user to implement many parts of the custom logic that is required to build state from windows and coordinate each parallel pass of an algorithm. For example, in the context of graph processing, the algorithms presented before exhibit many commonalities, such as the fact that we receive several edges or vertices in a data stream, create a graph state on a window and then run an iterative process at the granularity of each vertex. In this section, we present an example of a vertex-centric graph programming library that is built on top (currently integrated to Flink's graph streaming library [4])

[4]`https://github.com/vasia/gelly-streaming`

VertexContext Properties	**Definition**
vertexID: $\mathbb{N}$	The id of the current vertex
superstep: $\mathbb{N}$	The current superstep of the global computation
snapshotID: $\mathbb{N}$	The event time of the graph creation (window)
snapshotState: VertexState	The vertex state for the current graph snapshot
persistentState: VertexState	The persistent vertex state
isSource: Boolean	Flag for vertices that have no input edges
isSink: Boolean	Flag for vertices that have no output edges
neighbors: List[GraphVertex]	List of accessible graph vertices

Table 6.5: Variables Accessible under the Vertex Context

6.4.2.1 Vertex-Centric Iterations on Graph Snapshots

Pregel's vertex centric model has been widely adopted for declaring and executing distributed graph algorithms. Computation in the vertex-centric model is scoped per vertex and follows a bulk iterative synchronous execution (see section 6.2). The original model assumes a pre-built underlying static graph. For data streaming, a graph can only be built incrementally using streams of graph operations such as edge or vertex additions. Our vertex-centric model combines the concept of graph streams and static graphs via the following set of abstract types: `EdgeStream`, `VertexStream` and `GraphSnapshot`. An `EdgeStream` contains directed edges with two vertices and an optional custom state. Alternatively, the `VertexStream` contains vertices along with their neighbors and optional custom state.

Either vertex or edge streams can be used to define a `GraphSnapshot`, that is a full graph derived over a stream window of vertex or edge additions. Graph snapshots support fixed and fixpoint iterative computation summarized within a main compute function $(VertexContext, [Message]) \rightarrow (VertexContext, [Message])$ that accesses each vertex state (per-snapshot or persistent) as well as a set of other variables within the context of each vertex such as its list of neighbors, as summarized in Table 6.5.

Example: In Listing 6.5, we show how vertex-centric fixpoint iterations can be used to compute the length of the shortest path from a set of source nodes, a variant of the single source shortest path (SSSP) algorithm [121]. In this example, we consider the shortest path from nodes that have no input edges in the graph. Initially, we start with an `EdgeStream` for which we create a graph snapshot every 5 minutes. The "Direction.DIRECTED" parameter maintains the original direction of the graph edges in the snapshot (as opposed to "undericted" which includes the inverse edge in the graph snapshot). Within the first superstep, the source vertices store the zero length path to themselves and broadcast to their neighboring (target) vertices that value incremented by one. Then on each additional superstep, each vertex updates its local estimation of the shortest path by choosing the minimum of all the values received so far and further propagates that value incremented by one if a better estimation is found. The computation ends when all minimum paths have been computed and as a result, no more messages are sent further. Finally, each graph snapshot is converted back to a vertex stream which contains each vertex along with its minimum path from the sources.

The vertex-centric iteration model is a simple proof of concept of a programming model that is rather simple to use, requiring only a single function to be implemented for implementing graph algorithms. Its functionality is implemented fully using window iterations as such: The logic for the graph snapshot construction ("snapshot" call) happens within the entry function, the vertex computation within a step function and the conversion to an output vertex (toVertexStream) or edge stream (toEdgeStream) using the finalize function of a window iteration.

Listing 6.5: Single Source Shortest Path: Example of the Vertex-Centric API

```
//Stream of edges with nil (empty) state and vertices with 'Long' state
val input: EdgeStream<Long,Nil> edges = getEdgeStream()
//iterations on directed graph snapshots defined over a 5min tumbling window
input
  .snapshot(Minutes(5), Direction.DIRECTED)
  .runFixpoint(
  (ctx:VertexContext) => //VERTEX COMPUTATION
        {  //vertex computation
            if(ctx.superstep == 0 && ctx.isSource()){
                ctx.setSnapshotState(0l)
                ctx.neighbors.foreach(vertex => vertex.send(Message(1l)))
            }
            else{
                dist = ctx.getVertexState().getOrElse(Long.max)
                newDist = (dist :: ctx.getMessages().values).min
                if(dist != newDist){
                    ctx.setSnapshotState(distance);
                    ctx.neighbors.foreach(vertex => vertex.send(Message(newDist+1)))
                }
            }
        });
    .toVertexStream(); // creates a DataStream<GraphVertex(distance)>
```

6.4.3 Implementation Highlights

In this section, we highlight a few important parts of the implementation of the window iteration model. First, we show how the window iterate operator is composed on top of structured loops. Then we cover an important technical aspect of the implementation which is its sustainable flow control mechanism to eliminate cyclic deadlocks caused by Flink's default push-pull flow control mechanism. Finally, we discuss the integration of the window iterate operator with consistent managed state.

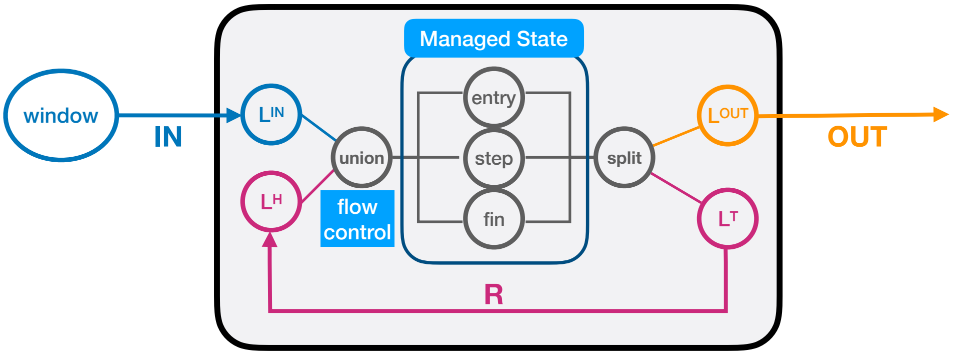


Figure 6.13: Design of the Window Iteration Operator in a Structured Loop.

6.4.3.1 Stream Graph Embeddings

In Section 6.3.3 we covered the graph embeddings of general structured loops. We will now examine how the window iteration operator is implemented within a structured loop in Apache Flink. Our overall design is depicted in Figure 6.13. There are three core operators (executed in parallel task instances): 1) A union of input and feedback streams, 2) The iterate operator that executes the user-defined functions in a sequential, thread-safe fashion and 3) a split operator that redirects output to the right data channels. Below, we will provide a short description of each of the operators

The *union* operator subscribes to both the input and feedback streams and its purpose is two-fold : 1) Flow control within cycles demands prioritization [122] and thus, an intermediate operator that consumes both input streams can enforce a respective strategy in a convenient way. 2) It simplifies the logic of the window iterate operator, having only one merged stream to operate (i.e., corresponding to a common type `Either[IN,R]`).

The *iterate* operator runs the core logic of the window iterations, as described before in subsection 6.4.1. For each input record or watermark, the iterate operator makes sure that the right user-defined function is invoked. There are several reasons why we chose not to split the iteration logic into multiple operators. First, managed state in Apache Flink can only be shared between operators in the same physical task. By collapsing all logic together we can provide access to the same context variables and managed state. Subsequently, it is important to guarantee thread safety, granting exclusive write and read access to managed state only for a single thread at a time. We can only achieve both of these requirements by enforcing sequential access to state across all of the three core functions (entry, step and finalize).

Finally, the *split* operator takes the combined output of the iterate operator and redirects it further to the right special operator (i.e., L^T and L^{OUT}).

6.4.3.2 Cyclic Flow Control

The default flow control mechanism in Apache Flink employs a form of natural backpressure. Tasks push and pull messages asynchronously across their message queues while leasing network buffers (fixed size). When no message buffer is available to deserialize an input message, consumption stops temporarily until network slots are freed once again. As a natural consequence, the physical network can potentially end up reaching its full capacity which results into producers blocking while trying to send. This process can propagate back to the stream sources and stream production halts until progress can be made. This strategy works adequately well for acyclic process graphs. However, cycles introduce a deadlock situation when network resources within structured loops are depleted. The default blocking mechanism would propagate back to the loop head and entry tasks to halt for an unsatisfiable condition.

In order to solve the problem of cyclic flow control and deadlocks it is important to make sure that cyclic computation can always make progress, while, external streams (going into a structured loop) halt until more network resources are available. The binary *union* operator can regulate traffic within its input channels given that it subscribes to both streams. An aggressive but safe strategy is to always prioritize traffic coming from the feedback stream. This way all resources within a structured loop will be fully allocated to complete an iterative process before ingesting the input for next one. Our employed strategy combines the safety of the aggressive approach while allowing a higher amount of multiplexing and it works as follows: Given $n \in \mathbb{N}$, the number of all network buffers available, we select a safety threshold of $k \in \mathbb{N}$ buffers where $k << n$. Initially both input and feedback streams have equal priority for ingestion. However, when the number of allocated buffers exceeds the threshold we switch to an aggressive strategy, where solely records from the feedback streams are being consumed. The threshold can also be seen as a variable that controls the degree of multiplexing within a structured loop, from low multiplexing ($k = 0$) to high ($k\ n$).

6.4.3.3 Managed State and Blocking Execution

Scalable managed state in Apache Flink is partitioned by key and managed by a certain operator instance during an execution (chapter 4). The iterate operator has full control of the state of every active iterative process and its execution logic is similar to that of stream windows. The entry or step function logic is executed when a progress condition is satisfied. In the case of the entry operator, that logic is executed once a new progress metric is notified (at the reception of a watermark). While new events of a certain context are entering the structured loop, the iterate operator stores those events until a watermark carrying that context is seen. This works as a signal mechanism for a completely ingested window and therefore, the entry function is invoked for that context. For every distinct context the iterate operator creates a new, dedicated managed state entry. Each iteration step works in a similar fashion and the step function has access to the same managed state entries as the entry function. Its execution logic resembles that of a stream window with a slide unit of 1 step, only here we have one instance of that window per context. All feedback records of a certain iteration step are collected until the progress metric indicates that this step is complete. Then, the step function is invoked on that input to generate new messages and state within the same context. The termination function is invoked once the loop head tasks indicates that via the use of termination watermarks. After the termination function is complete, the iterate operator purges the state of that context since it is no longer active. Both persistent and purgeable states are scalable and committed consistently for external use as described in Chapters 3 and 4. The same applies for the records that being collected before a computation is triggered in a step, since those are part of the managed state of the iterate task.

6.5 Limitations and Future Work

We presented structured loops and low watermarking as a solution to the problem of composing iterative processes on streams. We consider window iterations as a good starting point for richer programming models such as the complete integration of batch and stream programming models in Apache Flink. Furthermore, with new system capabilities we gain more opportunities for optimization. Below, we cover some of the limitations of the current approach as well as optimization and extension considerations for future work.

State-Based Termination: One missing, yet feasible functionality of iterative processes is the ability to define termination criteria based on the state transitions that occur within a step. So far we have shown how termination can be inferred via messages exchanged. However, it is possible to implement that logic via local decision making (i.e., within each iterate operator task) and the dissemination of special termination events from the iterate operator. In that case, the loop head would just work as an aggregator that makes sure that all local computation is complete before issuing the final termination protocol for a context.

Incremental and Shared Aggregation: The step function runs the core logic of an iterative process executed on each step, yet it is triggered on the complete input of an iteration rather than applied incrementally per record (see window partial aggregation in chapter 5). We did not focus on incremental aggregation in this chapter, since that is an orthogonal issue, to avoid confusion. Similarly to stream windows, it is possible to introduce partial aggregation scoped by context and iteration step. In that case the step function would process only the final result of the partial aggregation of a certain superstep.

Compatibility with Batch Programming: Apache Flink currently contains two distinct code bases and related shallow-embedded DSLs that make use of the same runtime, split into the batch (DataSet) and stream (DataStream) programming libraries. While its DataStream model is rich and can subsume the functionality of most of the existent batch workloads it has been lacking the ability to incorporate all the iterative processing capabilities of the the DataSet model. We therefore foresee the use of structured loops and window iterations as a good starting point to make such an integration feasible and allow for a more unified programming model for both types of processing in the future.

Performance Analysis: This chapter does not contain a performance analysis or comparison with another system. One of the core reasons is that there is no standard benchmark for iterative stream processing, given that it is a rather new concept. Furthermore, it is hard to distill the performance of core mechanisms for microbenchmarking (e.g., progress tracking) across different execution environments. However, a performance analysis will be highly important in the future to measure the capabilities of our decentralized long-running scheme compared to blocking coordination-driven execution (e.g., in Apache Spark or Batch Flink iterations) or Naiad/Timely Dataflow's global progress tracking mechanism.

6.6 Acknowledgments

Iterative processing had been an ongoing need since the conception of Gelly-Streams, a graph stream processing library developed as a joint effort with Vasiliki Kalavri and Daniel Bali, the feedback of which has been highly influential to the design of the primitives presented in this chapter. Furthermore, we would like to express our gratitude to Marius Melzer, who worked on the first proof-of-concept implementation of context-based watermarking and Flink's iterative processing framework during his internship in our team and MSc thesis work co-supervised with Pramod Bhatotia from TU Dresden (currently faculty at University of Edinburg). Finally, we would like to thank Frank McSherry, co-author of Naiad, for his insightful comments about Timely Dataflow and iterative stream progress tracking in general, a concept that originated from his team at Microsoft System Research (MSR) group back in 2013.

6.7 Summary: A Design Approach Perspective

In this chapter, we examined the problem of providing support for iterative processes on stream processing systems. We first provided a core model and primitives that describe an iterative process, such as a step function, termination criterion and the state of the computation. We then examined the implementation concerns in data-parallel systems such as global progress tracking and communication between process instances. We defined a partial order metric that is sufficient for tracking the progress of iterative processes in a stream graph and presented an extension of low watermarking that can be used to implement it. With the use of decentralized progress tracking we then showed how we can support structured loops on data streams as well as high level models that can be built on top to support iterative aggregation on stream windows. Our solution satisfies all target design principles and integrates well with all existing mechanisms present in data stream processing such as application-time watermarking, window aggregation and managed state.Below, we summarize how each of our design goals are satisfied within our proposed solution:

[D1] Blocking-Coordination Avoidance: Structured loops and context-based low watermarking provide a fully decentralized solution to tracking iterative progress. That is, no centralized or blocking-coordination protocol is necessary to inspect progress updates within the stream graph. In essence, with progress-based low watermarking we solved the problem of inferring a global progress condition with solely local information and decision making while computation progresses in an out-of-order fashion.

[D2] Runtime Transparency: Structured loops implement most of the necessary logic needed to maintain a cyclic stream computation via the use of their embedded operators. In fact, the programmer that uses structured loops or even the high level models for window iterations and vertex-centric computation is not exposed to these low level execution primitives. Iterative computation is abstracted through a small set of functions that are used for declaring the application-related parts of the computation (entry, step and finalization) rather than event-based logic.

[D3] Model Compositionality: Structured loops are, in essence, a structured programming primitive for data streams that supports arbitrary nesting and scoping. Effectively, we started from an arbitrary, unstructured programming model that discourages compositionality and proposed a structured programming model that allows for many different higher level models to be consistently built on top. The window iteration and vertex-centric libraries showcase several of these capabilities.

CHAPTER 7

Conclusions

We have examined the case of the stream processor as a general-purpose compute platform that can support arbitrary persistent state-driven workloads. The stream processing paradigm poses an attractive means of declaring persistent application logic with state over evolving data, allowing the composition of reliable services and complex applications beyond scalable data analytics which has been a major trend in the past decade. Prior work in the context of data streaming [123, 4, 2, 3, 124] originates from the fields of databases and data management with most significant emphasis put on rich semantics (e.g., windowing) rather than reliable and scalable execution. Throughout this work, we have revisited data stream processing to provide a clear system execution model via addressing a set of fundamental open system challenges: reliable stateful execution, computation sharing for sliding windows and support for structured iterations. In addition, we targeted for approaches that adhere to clear design goals: coordination avoidance, compositionality and programming model transparency.

The problem of reliable stateful execution relates to processing guarantees provided by a stream processing system at the presence of partial failures or reconfiguration needs and has been one of the biggest challenges so far. Our approach, models a distributed long-running execution into a series of phases called epochs, each of which must complete and commit changes atomically. In order to commit states consistently across epochs without coordination we make use of asynchronous epoch-cuts, a restricted case of Chandy and Lamport's consistent cuts, aiming to employ lightweight application-wide state snapshots after predefined execution epochs. Snapshots of epoch cuts can serve as the cornerstone of building complex application provenance dependencies and allowing application migration and reconfiguration that go beyond the need for fault tolerance, as it has been showcased in the Apache Flink system.

Computation sharing is especially important to long-running task execution, since it can lead to high savings in used computational resources for unbounded periods of time. Sliding windows in data streaming exhibit great potential for sharing computation and state, however, existing optimisations either make strong assumptions on the semantics of the windows or fall back to excessive over-consumption of resources to support semantic-agnostic execution sharing. Instead, in this work we examined the minimal set of semantics needed by a stream processor to achieve a best-effort computation sharing in sliding windows. The result was Cutty, an aggregation framework for compositional user-defined windows which can yield orders of magnitude of performance benefits when executing overlapping window computation compare to specialized state-of-the-art sharing computation techniques while maintaining a minimal memory footprint.

Finally, our study concludes with a thorough proposal of incorporating structured programming primitives to data streaming for iterative processing. Similarly to imperative programming, there has been a need for scopes and a strictly compositional control flow across different tasks in a stream processing graph. To that end, we propose the use of structured loops, a low level primitive for encapsulating subgraphs with feedback edges. Structured loops provide built-in support for iterative progress tracking as a special case of out-of-order processing using an extension of decentralized low-watermarking to infer iterative progress and thus, eliminating the need for a centralized or intermediate coordination mechanism to notify progress. Furthermore, we show how iterative processes can be composed using structured loops as an extension of window aggregation for multiple passes as well as higher-level programming abstractions such as the vertex-centric graph programming model.

Bibliography

[1] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 15, 2012.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *VLDBJ*, 2003.

[3] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The stanford data stream management system," *Book chapter*, 2004.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: continuous dataflow processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 668–668.

[5] "Apache Storm," http://storm.apache.org/, 2017.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2002, pp. 1–16.

[7] "The Lambda Architecture," http://lambda-architecture.net/, 2017.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] 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 *VLDB*, 2013.

[10] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 49–60.

[11] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 725–736.

[12] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*. USENIX Association, 2012, pp. 10–10.

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD Record*, 2005.

[14] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *AMC SIGMOD*, 2006.

[15] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," in *VLDB*, 2015.

[16] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[18] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[19] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 1, no. 10.4, 2014, p. 3.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: a timely dataflow system," in *ACM SOSP*, 2013.

[21] 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," *VLDB*, 2015.

[22] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: a new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008.

[23] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2004, pp. 263–274.

[24] P. D. Bailis, "Coordination avoidance in distributed databases," Ph.D. dissertation, UC Berkeley, 2015.

[25] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[26] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm." in *USENIX Annual Technical Conference*, 2014, pp. 305–319.

[27] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, 2010.

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

[29] P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink: Consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, 2017.

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

[31] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016.

[32] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *VLDB*, 2004.

[33] "Blink: How Alibaba Uses Apache Flink," http://data-artisans.com/blink-flink-alibaba-search/, 2016.

[34] "AthenaX : Uber's stream processing platform on Flink," http://sf.flink-forward.org/kb_sessions/athenax-ubers-streaming-processing-platform-on-flink/.

[35] "Stream processing with Flink at Netflix," http://sf.flink-forward.org/kb_sessions/keynote-stream-processing-with-flink-at-netflix/, 2017.

[36] "StreamING models, how ING adds models at runtime to catch fraudsters," http://sf.flink-forward.org/kb_sessions/streaming-models-how-ing-adds-models-at-runtime-to-catch-fraudsters/, 2017.

[37] "Real-time monitoring with Flink, Kafka and HB," http://2016.flink-forward.org/kb_sessions/a-brief-history-of-time-with-apache-flink-real-time-monitoring-and-analysis-with-flink-kafka-hb/, 2017.

[38] "Windowing and State in Storm," https://community.hortonworks.com/articles/14171/windowing-and-state-checkpointing-in-apache-storm.html, 2017.

[39] "Apache Apex," https://apex.apache.org, 2017.

[40] "[spark-20928] continuous processing mode," https://issues.apache.org/jira/browse/SPARK-20928.

[41] "[Proposal Document] Snapshots for Beam," https://docs.google.com/document/d/1UWhnYPgui0gUYOsuGcCjLuoOUlGA4QaY91n8p3wz9MY/edit, 2018.

[42] "Apache Beam," https://beam.apache.org/, 2017.

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

[44] P. Carbone, G. E. Gévay, G. Hermann, A. Katsifodimos, J. Soto, V. Markl, and S. Haridi, "Large-scale data stream processing systems," in *Handbook of Big Data Technologies*. Springer, 2017, pp. 219–260.

[45] P. Carbone, A. Katsifodimos, and S. Haridi, "Stream window aggregation semantics and optimization," 2018.

[46] "Apache Flink," http://flink.apache.org/, 2018.

[47] "Rockdb," http://rocksdb.org/, 2017.

[48] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[49] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, 2011.

[50] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language."

[51] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing." NetDB, 2011.

[52] "Apache Kafka project," http://kafka.apache.org/, 2017.

[53] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 46, 2014.

[54] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *ACM Sigplan Notices*. ACM, 2010.

[55] "Google Cloud Dataflow," https://cloud.google.com/dataflow/, 2017.

[56] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé *et al.*, "IBM Streams Processing Language: Analyzing big data in motion," *IBM Journal of Research and Development*, vol. 57, no. 3/4, pp. 7–1, 2013.

[57] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Nasgaard, R. Soule, and K. Wu, "SPL stream processing language specification," *NewYork: IBMResearchDivisionTJ. WatsonResearchCenter, IBM ResearchReport: RC24897 (W0911 044)*, 2009.

[58] A. Bifet and R. Gavalda, "Learning from time-changing data with adaptive windowing." in *SDM*. SIAM, 2007.

[59] D. Maier, J. Li, P. Tucker, K. Tufte, and V. Papadimos, "Semantics of data streams and operators," in *International Conference on Database Theory*. Springer, 2005, pp. 37–52.

[60] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[61] Y.-H. Feng, N.-F. Huang, and Y.-M. Wu, "Efficient and adaptive stateful replication for stream processing engines in high-availability cluster," *IEEE Transactions on Parallel & Distributed Systems*, no. 11, pp. 1788–1796, 2011.

[62] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the Borealis distributed stream processing system," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 1, p. 3, 2008.

[63] M. Balazinska, J.-H. Hwang, and M. A. Shah, "Fault-tolerance and high availability in data stream management systems," in *Encyclopedia of Database Systems*. Springer, 2009, pp. 1109–1115.

[64] 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," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1341–1352, 2016.

[65] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, no. 3, pp. 204–226, 1985.

[66] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[67] T. H. Lai and T. H. Yang, "On distributed snapshots," *Information Processing Letters*, vol. 25, no. 3, pp. 153–158, 1987.

[68] R. Guerraoui and L. Rodrigues, *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[69] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *ACM SIGMOD*, 2015.

[70] "Pravega," http://pravega.io/.

[71] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *Data Engineering, 2003. Proceedings. 19th International Conference on*. IEEE, 2003, pp. 25–36.

[72] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[73] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper, "Analytics on fast data: Main-memory database systems versus modern streaming systems."

[74] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 601–613.

[75] R. Tarjan, "Depth-first search and linear graph algorithms," in *Switching and Automata Theory, 1971., 12th Annual Symposium on*. IEEE, 1971, pp. 114–121.

[76] S. S. Muchnick, *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[77] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX annual technical conference*, vol. 8, 2010, p. 9.

[78] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[79] "Rbea: Scalable Real-Time Analytics at King," https://techblog.king.com/rbea-scalable-real-time-analytics-king/, 2016.

[80] "Managing Large State in Apache Flink: An Intro to Incremental Checkpointing," https://flink.apache.org/features/2018/01/30/incremental-checkpointing.html, 2018.

[81] H. Wang, L.-S. Peh, E. Koukoumidis, S. Tao, and M. C. Chan, "Meteor shower: A reliable stream processing system for commodity data centers," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012.

[82] G. De Francisci Morales and A. Bifet, "Samoa: Scalable advanced massive online analysis," *The Journal of Machine Learning Research*, vol. 16, no. 1, pp. 149–153, 2015.

[83] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, "Comet: batched stream processing for data intensive distributed computing," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 63–74.

[84] "The Trident Stream Processing Programming Model," http://storm.apache.org/releases/0.10.0/Trident-tutorial.html, 2017.

[85] "Low Latency Continuous Processsing Mode in Structured Stream in Apache Spark 2.3.0," https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html, 2018.

[86] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *VLDBJ*, 2006.

[87] "Apache Samza project," http://samza.apache.org/, 2017.

[88] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," in *ACM SIGMOD*, 2005.

[89] B. Gedik, "Generic windowing support for extensible stream processing systems," *Software: Practice and Experience*, 2014.

[90] T. Grabs, R. Schindlauer, R. Krishnan, J. Goldstein, and R. Fernández, "Introducing microsoft streaminsight," Technical report, Tech. Rep., 2009.

[91] A. Bifet and R. Gavaldà, "Adaptive learning from evolving data streams," in *Advances in Intelligent Data Analysis VIII*. Springer, 2009, pp. 249–260.

[92] Y. Yu, P. K. Gunda, and M. Isard, "Distributed aggregation for data-parallel computing: interfaces and implementations," in *ACM SIGOPS*, 2009.

[93] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," in *ACM SIGMOD*, 2003.

[94] K. Patroumpas and T. Sellis, "Window specification over data streams," in *Current Trends in Database Technology–EDBT 2006*. Springer, 2006, pp. 445–464.

[95] Z. Jerzak, T. Heinze, M. Fehr, D. Gröber, R. Hartung, and N. Stojanovic, "The DEBS 2012 grand challenge," in *ACM DEBS*, 2012.

[96] J. Li, K. Tufte, D. Maier, and V. Papadimos, "Adaptwid: An adaptive, memory-efficient window aggregation implementation," *IEEE Internet Computing*, 2008.

[97] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues, "Slider: incremental sliding window analytics," in *Proceedings of the 15th International Middleware Conference*. ACM, 2014, pp. 61–72.

[98] "Apache calcite," https://calcite.apache.org/.

[99] "Apache Spark project," http://spark.apache.org/, 2017.

[100] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 285–296, 2010.

[101] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "imapreduce: A distributed computing framework for iterative computation," *Journal of Grid Computing*, vol. 10, no. 1, pp. 47–68, 2012.

[102] A. Iyer, L. E. Li, and I. Stoica, "CellIQ: real-time cellular network analytics at scale," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 309–322.

[103] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: taking the pulse of a fast-changing and connected world," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 85–98.

[104] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 1.

[105] B. Chandramouli, J. Goldstein, and D. Maier, "On-the-fly progress detection in iterative stream queries," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 241–252, 2009.

[106] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi, "Incremental, iterative data processing with timely dataflow," *Communications of the ACM*, vol. 59, no. 10, pp. 75–83, 2016.

[107] "Introduction to Spark's Structured Streaming," https://www.oreilly.com/learning/apache-spark-2-0--introduction-to-structured-streaming, 2016.

[108] "Introduction to Kafka Streams," http://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple, 2017.

[109] M. A. Inda and R. H. Bisseling, "A simple and efficient parallel fft algorithm using the bsp model," *Parallel Computing*, vol. 27, no. 14, pp. 1847–1878, 2001.

[110] "Apache Hadoop project," https://hadoop.apache.org/, 2017.

[111] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[112] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer networks and ISDN systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[113] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[114] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.

[115] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM*

international symposium on high performance distributed computing. ACM, 2010, pp. 810–818.

[116] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "Ciel: a universal execution engine for distributed data-flow computing," in *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[117] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[118] X. Shi, B. Cui, Y. Shao, and Y. Tong, "Tornado: A system for real-time iterative analysis over evolving data," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 417–430.

[119] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl, "Spinning fast iterative data flows," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1268–1279, 2012.

[120] J.-M. Helary, C. Jard, N. Plouzeau, and M. Raynal, "Detection of stable properties in distributed applications," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, 1987, pp. 125–136.

[121] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2005, pp. 156–165.

[122] A. Lattuada, F. McSherry, and Z. Chothia, "Faucet: a user-level, modular technique for flow control in dataflow engines," in *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM, 2016, p. 2.

[123] 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.

[124] 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*, 2005.