DISS. ETH NO. 28957

You may not need synchronization (in streaming systems)

A thesis submitted to attain the degree of

Doctor of Sciences of ETH Zurich

(Dr. sc. ETH Zurich)

presented by

Andrea Lattuada

Master of Science ETH in Computer Science, ETH Zurich

born on 21. April 1989

citizen of Italy

accepted on the recommendation of

Prof. Dr. Timothy Roscoe (ETH Zurich), examiner
Prof. Dr. David Basin (ETH Zurich), co-examiner
Dr. Jonathan Mace (Max Planck Institute for Software Systems – MPI-SWS), co-examiner

2022

You may not need synchronization (in streaming systems).

Sommario

I sistemi moderni per l'analisi e elaborazione di dati fanno sempre più affidamento su sistemi distribuiti su rack e cluster per essere in grado di gestire la grande quantità di dati in ingresso e memoria richiesti, quando una solo nodo di computazione non è sufficiente. L'infrastruttura per mettere in produzione questi sistemi ha un costo elevato e quindi ogni miglioramento dell'efficienza ha come risultato un significativo risparmio di risorse, e permette di intraprendere analisi più sofisticate. Allo stesso tempo le astrazioni per le programmazione usate per l'elaborazione di dati in parallelo che supportano l'abilità di scalare l'esecuzione in modo semi-automatico richiedono un significativo compromesso a livello di prestazioni del sistema; il costo di queste astrazioni può superare i benefici portati dalla scalabilità orizzontale.

I modelli di programmazione dataflow con coordinamento dettagliato (fine-grained) basato su epoche sono stati sviluppati per avere un basso overhead: i sistemi basati su questi modelli permettono di implementare molti programmi di analisi ed elaborazioni di dati in modo efficiente. Tuttavia i meccanismi di sistema come il re-scaling dinaminco, il ri-partizionamento on-line di dati, la fault-tolerance (tolleranza alle anomalie) e la condivisione degli indici devono tutti essere adattati in modo che siano compatibili con il più ocmplesso modello di esecuzione e che introducano il minor overhead possibile, per evitare di sperperare la maggiore efficienza di questi sistemi. Tali meccanismi sono spesso necessari per mettere in produzione questi sistemi nel mondo reale.

Questa tesi descrive come adattare il modello di programmazione dataflow per permettere di implementare di meccanismi di condivisione degli indici, ri-partizionamento, re-scaling, fault-tolerance, e gestione delle risorse in forma di librerie opzionali scritte direttamente utilizzando il sistema dataflow di base, il quale deve solo fornire le giuste primitive dataflow. La tesi poi dimostra come construire tali meccanismi con un overhead di througput accettabile e cono un costo a livello di latenza predicibile e limitato, in modo tale che essi siano utilizzabili in applicazioni interattive.

Presentiamo una nuova astrazione per la programmazione di sitemi dataflow con parallelismo sui dati: una primitiva di coordinamento che il programma dataflow può usare per comunicare in modo preciso segnali di coordinamento dettagliati (fine-grained). Utilizzando questa astrazione abbiamo progettato e implementato un meccanismo per la condivisione degli indici ispirato dai sistemi di gestione delle basi dati (DBMS) che abbiamo adattato ai sistemi dataflow distribuiti usando il coodinamento fine-grained. Abbiamo inoltre realizzato protocolli di fault-tolerance e re-scaling dinamico con prestazioni predicibili. Allo scopo di assicurare la correttezza di questi meccanismi e dei programmi di elaborazione dati, abbiamo anche formalizzato e verificato il protocollo di coordinamento principale di uno stream processor allo stato dell'arte, che supporta la nostra nuova primitiva di coordinamento.

Abstract

Modern data analytics and processing systems are increasingly relying on rack-scale or cluster-scale systems to deal with massive input rates and memory requirements that cannot be handled by a single compute node. The infrastructure to run these systems has an high cost: gains in efficiency result in big savings, and enable more sophisticated analyses. At the same time the programming abstractions used for data-parallel data processing that provide semi-automating scaling come with a significant performance tradeoff, and the cost of abstraction can often outweigh the performance gains due to horizontal scaling.

Dataflow programming models with epoch-based, fine-grained coordination were developed to have significantly less intrinsic overhead: systems based on these models enable the efficient implementation of many large-scale low-latency data analytics and processing tasks. However, system mechanisms such as dynamic re-scaling, on-line data re-partitioning, fault-tolerance, and index sharing need to be adapted to cope with the more complex execution model, and need to introduce minimal overhead, to avoid squandering these systems' increased efficiency. These mechanisms are often necessary to deploy these systems in the real world.

This thesis describes how to adapt the distributed dataflow programming model to enable the implementation of low-overhead, predictable index sharing, re-scaling, re-partitioning, fault tolerance and resource management systems as optional libraries written against the core dataflow system that only needs to provide dataflow primitives. It then demonstrates how to build these mechanisms with acceptable throughput overhead and predictable, bounded latency cost, so they are suitable for interactive applications.

We present a new programming abstraction for data-parallel dataflow systems: a coordination primitive that the dataflow program can use to precisely signal fine-grained coordination information. Building on this abstraction, we design and implement a data index sharing mechanism inspired by DBMSes and adapted to distributed dataflow systems with fine-grained coordination and a fault-tolerance and dynamic re-scaling protocol with predictable performance. To help ensure correctness of these mechanisms and of the data processing tasks, we also formalize and verify the core coordination protocol of a state-of-the-art stream processor that supports our new coordination primitive.

A Mamma, Papà, Fraise
Non ce l'avrei fatta senza di voi

To Mom, Dad, Fraise
I couldn't have done it without you

Acknowledgements

This dissertation would not exist without the involvement of many people.

I am grateful to Prof. Mothy Roscoe for first giving me the opportunity to work in the Systems Group and, more importantly, for believing in my ability to be successful as a PhD student, even when more than once I doubted that myself.

I would also like to thank Prof. David Basin and Dr. Jonathan Mace for being part of my doctorate committee.

I owe a lot of my research style, philosophy, and (hopefully) integrity to Dr. Frank McSherry; I am grateful he supported me and helped me with the projects that make up this dissertation. Working with Frank on my Master Thesis and PhD research has been an incredibly intellectually stimulating experience.

I am privileged to have been supported financially by a Google PhD Fellowship and I would like to thank Dennis Fetterly, my Google mentor, who was always excited to see where my changing interests would take my work and who listened to me when I inevitably encountered hard times during my studies.

I am deeply grateful to Dr. Jon Howell, for helping me find my way when due to circumstances I had to make significant changes to my research direction. Thank you Jon for two great summer internships, for all your invaluable feedback, and for your friendship. I would also like to thank Rob Johnson, who was an excellent co-mentor alongside Jon at VMware Research: I'm looking forward to more sailing together.

A lot of the work in this dissertation would not be the same without the involvement of the ETH Master students I had the privilege to work with. Thank you Matthias Brun, Sára Decova, Lorenzo Martini, and Lorenzo Selvatici. And thank you to the Systems Group summer intern Isitha Subasinghe. You are all incredible.

I am also grateful to my ETH and external collaborators; in particular thank you Sebastian Wicki and Moritz Hoffmann for your work on *Snail-trail* and *Megaphone*, thank you Dr. Malte Schwarzkopf for the seemingly endless revision work on our paper, and thank you Dr. Dmitriy Traytel for teaching me some Isabelle and for your work on our paper.

Thank you Dr. Chris Hawblitzel for embarking with me on the journey to build Verus; this last year and a half has been a blast.

During my PhD studies I have been privileged to work with Travis Hance, Chanhee Cho, Yi Zhou, Jay Bosamiya, and Bryan Parno at Carnegie Mellon University; Chris Hawblitzel at Microsoft Research; Jialin Li at University of Washington; Gerd Zellweger, Alex Conway, and Jon Howell at VMWare Research; Ryan Stutsman at the University of Utah. The research work I have done with you all is not part of this dissertation but you are all incredible people to work with.

It takes a village of unusual people to grow a Doctor. Thank you to all my friends and colleagues at the ETH Systems Group: Abishek, Anastasiia, Ben, Cédric, Claude, Dan, Daniel, Dario, David, Dimitris, Fabio, Gerd, Hidde, Ingo, Kaan, Kornilios, Lazar, Lucas, Lukas, Maurice, Melissa, Merve, Michal, Michal [sic.], Moritz, Nicolas, Nora, Pravin, Renato, Reto, Roni, Simon, Simon [sic.], Tom, Vojislav, Zhenhao, Zsolt. Thank you Michael for your moral support in these last few months. And of course thank you to Ana, Ce, Gustavo, and Theo.

Thank you Dr. Ghislain Fourny for being an excellent lecturer for many of the courses I had the pleasure to help with as a Teaching Assistant. And a big thank you to the System Group's world class admin team: Jena, Nadia, and Simonetta.

And if you made it here, you are either bored or one of my beautiful friends who shared the good, and the bad times with me: Benni, Chicca, Cianciu, Dani, David, Dimitris, Donz, Forte, Frenk, Frenzo, Gallo, Georg, Grizzo, Malte, Mau, Michi, Michael, Niko, Renato, Rik, Ste, Stefano, Stephan, Tabo, and also you, that somehow I forgot, I owe you a Negroni. I don't say this much, but you're the best, really.

Elda e Turi, thank you for supporting me through all my lows, for never doubting I could do it, for always being proud of me, for helping me see the bright side, and for being the best dog-family for Fraise. You can't read this, but thank you Fraise, you got me through the worst times.

Contents

1 Introduction

Modern data analytics and processing systems are increasingly relying on rack-scale or cluster-scale systems to deal with massive data input rates and memory requirements that cannot be handled by a single compute node. The infrastructure to run these systems has an high cost, thus gains in efficiency result in big savings, and enable more sophisticated analyses.

Distributed streaming and data processing systems provide abstraction to make it simpler to write scale-out analyses and queries over large inputs. This semi-automated scalability comes with a cost. [MIM15] Many of these systems are far from achieving the performance of a well-tuned program written without the help of those abstractions, as the cost of abstraction can often outweigh the performance gains due to horizontal scaling. Nevertheless, the popularity of these systems indicates that such abstractions enable rapid development, are reliable, and easier to understand for the programmer.

The MapReduce programming model [DG04] popularized scale-out distributed data processing: in this model users "specify the computation in terms of a *map* and *reduce* function" which are automatically parallelized by a runtime system. Dryad [Isa+07] generalized MapReduce to a dataflow structure where generalized *map*/*reduce* stages are chained together to encode more complex tasks. Popular streaming dataflow systems combine this scale-out approach with the streaming programming model of stream processing engines [Aba+03; Bal+05] and database query processing [Gra94] to handle high-volume low-latency queries over streaming inputs.

A streaming dataflow program is represented as a graph of such inter-

connected operators that perform per-record data transformations. Like MapReduce, this model enables automatic and seamless parallelization of streaming tasks on large multiprocessor systems, and cluster-scale deployments. Many research-oriented and industry-grade systems have employed this model to describe streaming transformations and aggregation for large-scale big-data analytics, on-line analytics processing, and machine learning tasks. In addition, analysts and engineers often require stream processing systems to also provide millisecond-scale latency guarantees at the 99th percentile [DB13]: scale-out dataflow systems need to provide good latency guarantees in order to be viable for these interactive applications.

System mechanisms such as dynamic re-scaling, on-line data re-partitioning, fault-tolerance, resource management, and index sharing are critical to deploying streaming dataflow systems in the real-world. For example, monitoring applications based on streaming dataflow systems need to recover quickly after a failure to provide timely alerts and automatic remediations for the monitored system.

Dataflow programming models with epoch-based, fine-grained coordination were developed to have significantly less intrinsic overhead [Mur+13]: systems based on these models enable the efficient implementation of many large-scale low-latency data analytics and processing tasks (Naiad [Mur+13] and Flink [Car+15] are among such systems). However, the aforementioned systems mechanisms must be adapted to cope with the more complex execution model, while introducing minimal overhead, to avoid squandering these systems' increased efficiency.

To the best of our knowledge mechanisms that pre-date this thesis were either entirely unsuitable for fine-grained streaming systems, imposed overheads that negated the performance gains of scaling out, or introduced latency overhead and variability that made these systems unsuitable for interactive applications.

This thesis describes how to adapt the distributed dataflow programming model to implement of low-overhead, predictable index sharing, re-scaling, re-partitioning, fault tolerance and resource management systems as optional libraries written against the core dataflow system that only needs to provide dataflow primitives. It then demonstrates how to build these mechanisms with acceptable throughput overhead and predictable, bounded latency cost, so they are suitable for interactive applications.

In modern time-aware dataflow systems records flow through the dataflow graph asynchronously, to enable efficiency through pipeline and task parallelism. Concurrent updates may race along the multiple paths between dataflow operators (potentially distributed across multiple threads of control) and arrive in different orders than they were produced. For dataflow operators to compute correct results, time-aware dataflow systems assign a logical timestamp to messages and exchange control signals to determine which timestamps can be retired at which operator in the graph.

This thesis proposes a new programming abstraction for these time-aware dataflow systems: a coordination primitive that dataflow operators use to explicitly signal which timestamps are in-progress or retired. This abstraction is sufficient to construct complex synchronization protocols like the ones necessary for mechanisms such as index sharing, data re-partitioning, and fault tolerance without modifying the core system.

This approach benefits from the ability to use timestamp to both (i) determine which timestamps can be retired at which operator and (ii) build the synchronization protocols for those system mechanisms. The ordering information implicitly encoded in timestamps make them the right signal to coordinate data re-partitioning, incremental indexing, and recording and recovering state across failures. This thesis discusses how to leverage this property of timestamps to implement these mechanisms with acceptable performance overhead.

By reusing the intrinsic ordering information of timestamps, system mechanisms do not need to introduce their own entirely separate and potentially expensive coordination protocols and control signals. They can instead rely on the existing coordination information that is already computed by the system for operators to compute correct results and they only need to induce the additional signals they require to accomplish their function (for example, consistently re-partitioning operator state on-line).

Because these mechanisms are now libraries and need not be integrated in the core system, an application can elect to include only the features that are required for the target deployment scenario: a monitoring application may require fault tolerance but may not need on-line data re-partitioning if the input data distribution remains stable over its lifetime. This modular design ensures that applications do not experience performance or complexity overhead associated with a system feature they do not need. This modularity is necessary to build "zero-cost abstractions":

system and language features that have no cost when they are not used, and have cost comparable to a carefully hand-coded solution otherwise.[1]

The refined dataflow abstraction this thesis introduces also enables a simpler system model that can be directly expressed in a formal language. This enabled formal verification of the correctness of the core coordination protocol of the most advanced time-aware dataflow system to-date, which is also the research vehicle for the projects discussed in this thesis. Formally verifying the coordination protocol helps to ensure the correctness of the results computed by dataflow programs running on the system, and of the fault-tolerance, indexing, and re-partitioning mechanisms, which rely on the coordination protocol.

[1] The concept of "zero-cost abstractions" as a design goal is generally attributed to Bjarne Stroustroup [Str95].

1.1 Overview

This thesis first introduces the dataflow programming model, from the perspective of prior work, in Chapter 2. Chapter 3 presents a new, generalized dataflow system and programming model that captures the semantics of modern distributed time-aware dataflow systems designed to process high-rate input streams with low latency.

Chapter 4 introduces a new coordination primitive for dataflow systems, the *timestamp token*, and a refined dataflow programming model that minimizes the volume of information shared between the computation and host system, without surrendering precision about concurrency. Dataflow operators can explicitly signal which timestamps are in-progress or retired and can build more complex synchronization protocols on the basis of this new primitive.

The following chapters, starting with Chapter 5, address operational and system concerns of dataflow systems: indexing, re-scaling, fault-tolerance, and flow control. In addition to independent contributions to addressing these concerns, these chapters also discuss how timestamp tokens allow programs and frameworks to abstractly but precisely express sophisticated coordination protocols using exclusively timestamp tokens and no additional coordination primitives.

Chapter 6 presents a new design for stream-based query processors that maintains indexed views of intermediate dataflow operator state. This shared state allows concurrent queries to reuse the same in memory state without compromising data-parallel performance and scaling. This design relies on timestamp tokens to encode its coordination mechanism.

Chapter 7 presents a new design for a fault-tolerance mechanism for time-aware dataflow that moves coordination off the critical path to avoid latency spikes when intermediate state is durably persisted. This mechanism is again implemented as a library that uses the timestamp tokens abstraction to coordinate the garbage collection of durable state in steady state and the recovery protocol.

Finally, timestamp tokens provide a cleaner formal model and definition of correctness for a time-aware dataflow system. Chapter 8 presents the model, and presents a machine-checked proof of the core coordination protocol of timely dataflow.

1.2 Notes on collaborative work

This thesis includes results from collaborations with:

Matthias Brun

Dr. Zaheer Chothia

Sára Decova

Dr. Desislava C. Dimitrova

Dr. Moritz Hoffmann

Dr. Vasiliki Kalavri

Dr. John Liagouris

Dr. Frank McSherry

Prof. Dr. Timothy Roscoe

Dr. Malte Schwarzkopf

Lorenzo Selvatici

Dr. Dmitriy Traytel

Sebastian Wicki

Some of the work presented in this dissertation has been published in some form:

[Bru+21] Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. "Verified Progress Tracking for Timely Dataflow". In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. 2021.

[LM22] Andrea Lattuada and Frank McSherry. *Timestamp tokens: a better coordination primitive for data-processing systems*. 2022.

[Lat16] Andrea Lattuada. "Programmable scheduling in a stream processing system". Masterarbeit. Systems Group, Department of Computer Science, ETH Zurich. 2015-2016. Nr. 144. Master Thesis. ETH Zurich, 2016.

[McS+20] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. "Shared Arrangements: practical inter-query sharing for streaming dataflows". In: *Proc. VLDB Endow.* 13.10 (2020).

[McS+18] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. *Shared Arrangements: practical inter-query sharing for streaming dataflows.* 2018.

During my PhD I co-supervised several Master Theses:

[Mar19] Lorenzo Martini. "Performance implications of a Dataflow System's Communication Plane". Master Thesis. ETH Zurich, 2019.

[Dec20] Sára Decova. "Modelling and Verification of the Timely Dataflow Progress Tracking Protocol". Master Thesis. ETH Zurich, 2020.

[Sel04] Lorenzo Selvatici. "A Streaming System with Coordination-Free Fault-Tolerance". MA thesis. ETH Zurich, 2020-04.

In particular:

- Chapter 3 is based in part on "Verified Progress Tracking for Timely Dataflow" [Bru+21], *Timestamp tokens: a better coordination primitive for data-processing systems* [LM22], and "Shared Arrangements: practical inter-query sharing for streaming dataflows" [McS+20];
- Chapter 4 and Chapter 5 are based on *Timestamp tokens: a better coordination primitive for data-processing systems* [LM22]; some of the work presented in Chapter 4 and Chapter 4 was initially reported as part of my Master Thesis "Programmable scheduling in a stream processing system" [Lat16];
- Chapter 6 is based on "Shared Arrangements: practical inter-query sharing for streaming dataflows" [McS+20] and the associated Technical Report [McS+18];
- parts of Chapter 7 describe joint work with Lorenzo Selvatici in the context of his Master Thesis "A Streaming System with Coordination-Free Fault-Tolerance" [Sel04]; Frank McSherry and Moritz Hoffmann contributed to an early prototype of the mechanism described in Chapter 7;

- Chapter 8 is based on "Verified Progress Tracking for Timely Dataflow" [Bru+21] and references work done by Sára Decova in the context of her Master Thesis "Modelling and Verification of the Timely Dataflow Progress Tracking Protocol";

Additionally, section 5.3 refers to joint work with the authors of "Megaphone: Latency-conscious state migration for distributed streaming dataflows" [Hof+19] which was included in the following ETH Doctoral Thesis:

[Hof19] Moritz Hoffmann. "Managing and understanding distributed stream processing". Doctoral Thesis. ETH Zurich, 2019.

2 Dataflow model

This chapter provides a high level intuition of the dataflow programming model and then introduces the properties and semantics of dataflow programming models described in prior literature.

In general, the dataflow model is designed to program data-processing applications as a collection of potentially parallel activities and their data-dependencies; this enable a runtime system to (semi-)automatically correctly schedule the execution of the tasks on multiple processors in parallel, enabling horizontal scale-out. What varies between different dataflow models is (i) whether they support one-shot batch jobs or continuous streaming queries, (ii) which flavors of available parallelism (and associated data dependencies) can be expressed, and (iii) the granularity at which such data dependencies can be expressed.

2.1 Dataflow graph and dataflow computation

Most dataflow model flavors we will consider have a basic common graph structure, an example of which is depicted in Figure 2.1.

A **dataflow graph** (V, E) consists of a set of vertices V and a relation $E : V \to V$ which represents the directed graph edges: an edge from $v_1 \in V$ to $v_2 \in V$ is represented as $(v_1, v_2) \in E$. In a **dataflow program**, a complete function $P : V \to C$ associates each vertex with some program behavior, generally specified as one or more fragments of sequential program code. In the context of data processing, the graph edges E are communication **channels** that transport data between the dataflow **op-**

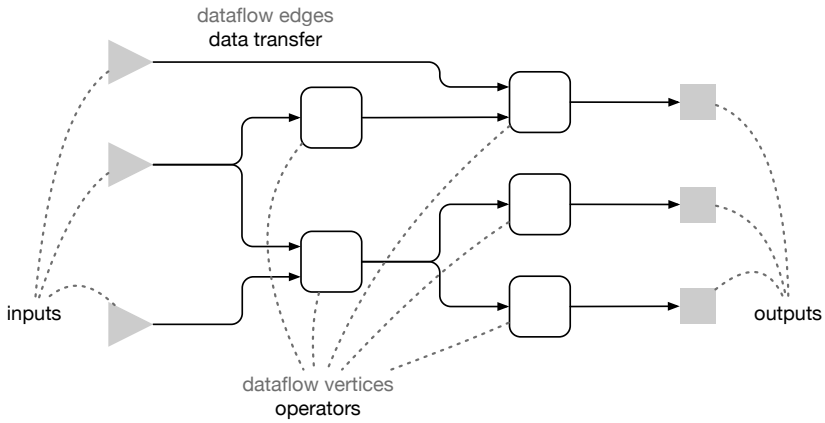


Figure 2.1: The parts of a dataflow graph representing a dataflow program.

erators, represented by the graph vertices. Each operator is associated with some data processing function or behavior (this is represented by the relation P above).

A **computation** over input data is written as a **dataflow program** where vertices with no incoming edges represent the input datasets ($V_{in} \subseteq V$) and vertices with no departing edges represent the computation's outputs ($V_{out} \subseteq V$). Each vertex (operator) performs a data transformation on the data received from its incoming edges (input channels) which it forwards via its departing edges (output channels) to the next operators (or the outputs).

Channels between operators that do not act as inputs or outputs carry intermediate computation results. These are represented by the edges $E_{inter} = \{(v1, v2) | v1 \notin V_{in} \wedge v2 \notin V_{out}\}$. As such, the dataflow edges represent *data dependencies* between the inputs, the operators, and the outputs. The transitive closure E^+ of the edges relation E captures the pair-wise direct or indirect data dependencies between inputs, operators, and outputs: a vertex v_x (output or operator) has a data dependency on any other operator or input v_d if $(v_d, v_x) \in E^+$. Figure 2.2 depicts the transitive data dependencies of the vertex representing one of the

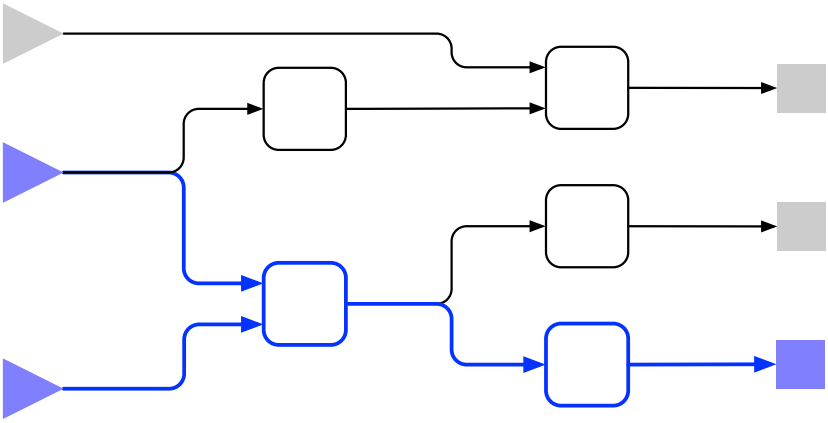


Figure 2.2: A dataflow graph with the transitive data dependencies of one of the output vertices highlighted.

computation outputs in the example graph.

In the following we will use these definitions to describe variations to the model even when they deviate from the terms used in their original formulation: when necessary, we will relate these definitions to the terms used in prior work. In figures, we will omit vertices representing inputs and outputs in the diagrams for clarity.

2.2 Example of a dataflow computation

Figure 2.3 shows a diagram of an example dataflow program for high-rate data analytics as seen in contemporary deployments. The example is adapted from the NEXMark benchmark suite [Tuc+08] for query processing systems over data streams: the scenario is a on-line auction house website (like eBay [eBa]) where information about auctions and bids are streamed into the data-processing system.

The dataflow program in the example computes and updates the results for three continuous queries: (1) the top-k categories by number of bids in

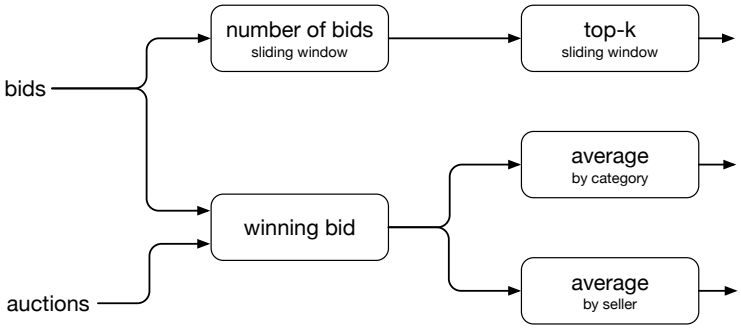


Figure 2.3: An example NEXMark-inspired dataflow computing the top-k categories by the recent number of bids, the running average winning bid for each category, and the running average winning bid for each seller.

the last time window (e.g. an hour), (2) the running average winning bid for each category, (3) and the running average winning bid for each seller. Bid and auction updates are streamed into the system, processed by the dataflow operators, and may result in updates to the query outputs that are streamed out of the system.

Different flavors of the dataflow model will encode this task in different ways, depending on their support for expressing parallelism, how data is transferred between operators, and their data models.

2.3 Batch and streaming inputs

There are two families of data processing tasks and systems where programs are represented as dataflow programs: (i) one-shot batch processing jobs, and (ii) continuous stream processing tasks or continuous queries.

2.3.1 Batch

MapReduce [DG04], Spark [Zah+12], and Dryad [Isa+07] are representative of programming models and systems designed for scale-out batch processing of large input datasets. These systems have been widely employed in the industry for querying and transforming large datasets for business intelligence, operational tasks, periodic and expensive data analyses (backing for example recommender or fraud detection systems), security and log analyses, indexing, and more.

For batch processing, the input vertices in a dataflow program are input datasets (generally as collections of records) and the dataflow operators represent transformations such as filtering, joins, and aggregations on the datasets identified by the operator's incoming edges. These data transformations are data-parallel, as described later in subsection 2.4.4.

Data dependencies In the batch model the dataflow edges correspond to data dependencies between the transformed datasets computed by the various operators (vertices) or the inputs. To complete the data transformation associated with an operator, all its transitive data dependencies must have been completed too.

2.3.2 Streaming

Apache Storm [Apa], Apache Spark Streaming [Zah+13], Naiad [Mur+13; MT], Timely dataflow [MT], Apache Flink [Car+15], and Google Cloud Dataflow [Aki+15] are representative of programming models and systems designed for scale-out stream processing over changing, high-volume input datasets often represented by high-rate streams of incoming "updates" (records). These updates originate from user operations (similarly to writes to a database), monitoring services (that track hardware metrics and events), sensors, and other continuous data sources.

These systems are widely deployed for developing micro-services which are components of interactive web or management services, such as system monitoring, intrusion detection, engagement metric aggregation ("view counts and likes"), service billing, online data analytics and business intelligence, and machine learning inference.

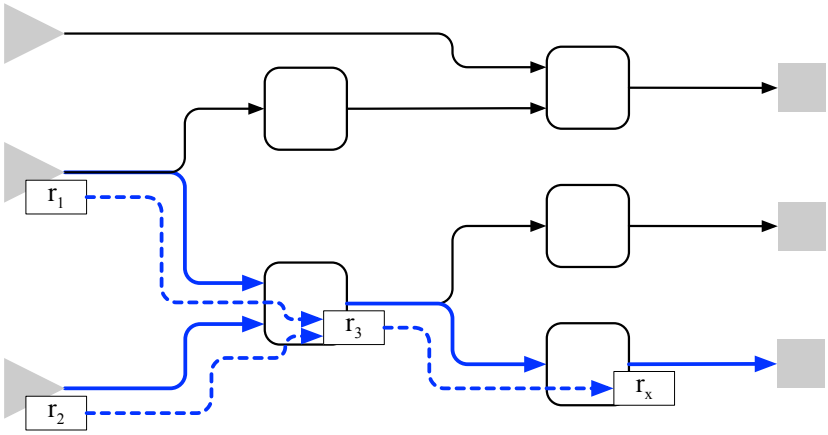


Figure 2.4: A dataflow graph with the transitive data dependencies of one of the output records highlighted. r_x depends on r_3 which in turn depends on r_1 and r_2.

Data dependencies In streaming systems the input vertices represent continuous streams of incoming records. The dataflow edges carry record streams between the dataflow operators which process them one record at a time (or one record batch at a time). The dataflow edges correspond to data dependencies between individual records as they appear on the output of the vertices. To complete the data transformation associated with an operator for a record r_x, all its transitive data dependencies must have completed processing of the intermediate or input records that resulted in r_x. Figure 2.4 depicts the exact transitive data dependencies of a record r_x that appears in the output. This dependency relation for records can be seen as a family of graphs whose edges E_r are subsets of the dataflow graph edge set E ($E_r \in E$).

Scale-out dataflow streaming systems aim to provide scalable performance to handle massive input streams but otherwise have similar design goals and use cases as traditional streaming data-processing engines and continuous query processors.

2.3.3 Data model

Record tuple Dataflow data processors inherit the data model of databases. Inputs, intermediate results and outputs are usually a collection of record tuples $(a_1, ..., a_n)$ although some systems support hierarchical or unstructured data as records (e.g. video frames). In batch systems these collections, once fully computed, are static. Following Aurora [Aba+03], these collections in streaming systems are append-only sequences of record tuples.

Record key Aurora [Çet+16] also introduces the concept of an optional tuple key for the stream. A tuple $(k_1, ..., k_m, a_1, ..., a_m)$ has a key $(k_1, ..., k_m)$ for the stream and attribute values $(a_1, ..., a_m)$. This key acts similarly to a key in a database table: it identifies records that refer to the same entity. For example, all record tuples referring to a certain user of a system will contain the user identifier as part of their key. This enables data aggregations by key, like counting all of a user's interaction with a system in a specified amount of time.

Timestamp In Aurora and many streaming systems since, each record tuple in a stream has a timestamp t that indicates when it originated: the timestamp can be a wall clock time of when the record tuple was generated or a logical value shared by records generated at the same logical time (for example a monotonic transaction counter). The timestamps of records increase over the life-cycle of the dataflow computation.

In Aurora and Borealis the timestamp was used exclusively for quality of service calculations and hidden from data transformation logic at operators. In modern time-aware systems the timestamp is used in data transformations for temporal aggregation (for example, for computing time-windowed statistics). This thesis will formally define timestamps in section 2.6 and will discuss the time-aware dataflow model in Chapter 3.

Revision Borealis and Differential dataflow [McS+13], among others, also support an extended model to revise the information encoded in a stream with insertion, deletion, and replacement records. The following definitions use a notation similar to the one introduced by Borealis; other systems that support revisions use variations of this model.

- An *insertion message* $(+, d)$ indicates that $d = (k_1, ..., k_m, a_1, ..., a_m)$ is a new record tuple to be inserted with key value $(k_1, ..., k_m)$;

- a *deletion message* $(-, d)$, where $d = (k_1, ..., k_m)$, removes the previously inserted record tuple with key value $(k_1, ..., k_m)$;

- a *replacement message* $(\leftarrow, d)$, where $d = (k_1, ..., k_m, a_1, ..., a_m)$, takes a previously inserted record tuple with key value $(k_1, ..., k_m)$ and revises its attribute values with $(a_1, ..., a_m)$.

2.4 Expressible parallelism

Stream processing systems that process high-rate streams use the dataflow programming model because it allows programmers to abstractly express parallelism opportunities in the programs, which these systems can exploit when scaling out. Figure 2.5 depicts three of the four kinds of parallelism that can be exploited in stream processing systems: task-, pipeline-, and data-parallelism. In addition, stream processing dataflow models that use timestamps can leverage another epoch parallelism, an other kind of intra-operator parallelism.

The rest of this chapter defines each form of available parallelism following the taxonomy introduced by Volcano [Gra94]. Volcano is an early dataflow-based parallel query processor that identified the types of parallelism expressed by the dataflow model (without timestamps).

Volcano distinguished and supported both push-based and 'demand-driven' dataflow: in push-based dataflow ('flow-control' in Volcano) operators can produce and "push" records on their own schedule, while in 'demand-driven' dataflow operators produce records in response to a request from a consumer (which then waits for their completion): in our dataflow model, producers and consumers are respectively the source and destination of a dataflow edge (note that all operators, except for inputs and outputs, acts as both producers and consumers for different channels). Almost all modern scale-out dataflow systems use the push-based dataflow model which will focus on in the following.

This section refers to Volcano's definitions for parallelism but uses terminology more commonly used in modern systems.

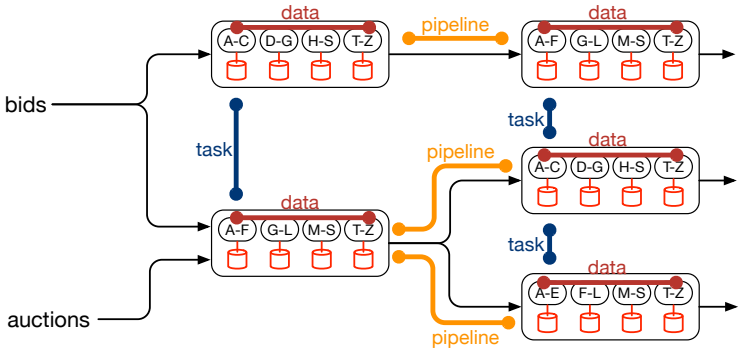


Figure 2.5: Available parallelism in a dataflow program.

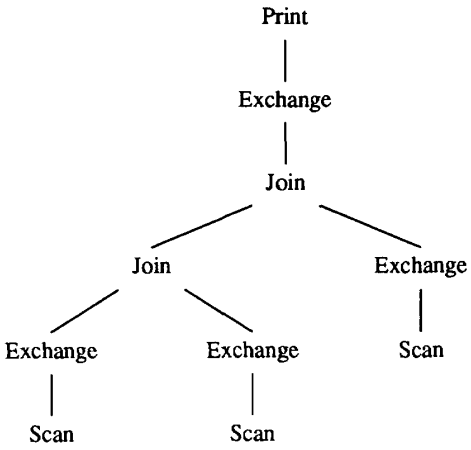


Figure 2.6: (Reproduced from [Gra94], page 130.) Model of operator parallelism in Volcano.

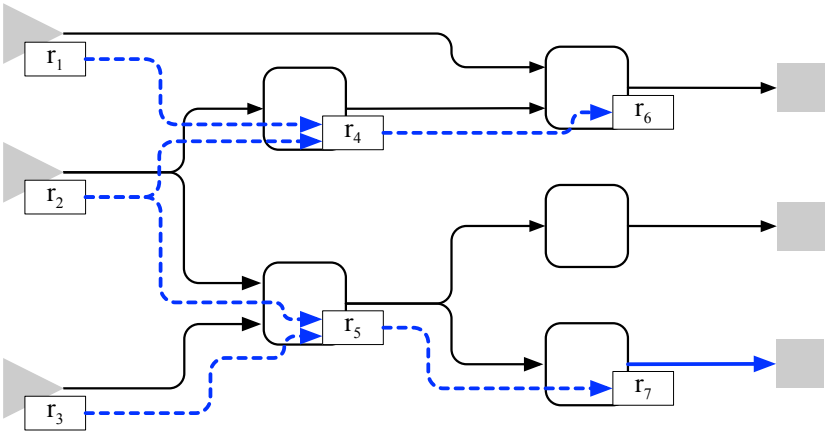


Figure 2.7: Example of task parallelism for records in a dataflow graph.

2.4.1 Task parallelism

Data transformations that do not have a data dependency on each other can be executed in parallel. Formally, the code fragments $P(v_1)$ and $P(v_2)$ associated with two operators v_1 and v_2 for which $(v_1, v_2) \notin E^+ \wedge (v_2, v_1) \notin E^+$ can be executed in parallel. This extends to streaming systems where data transformations on record tuples r_1 and r_2 for which $(r_1, r_2) \notin E_r^+ \wedge (r_2, r_1) \notin E_r^+$ can be performed in parallel.[1]

Figure 2.7 shows an example of available task parallelism in a streaming setting. Records r_4 and r_5 do not depend on each other and they can be constructed in parallel by their respective operators. The same is true for the pairs r_6, r_7, r_4, r_7, and r_6, r_5.

A similar concept is referred to as 'Bushy parallelism' in Volcano [Gra94] which defines it as different processors executing different subgraph of a complex dataflow graph.

[1] As a reminder: $P : V \rightarrow C$ associates each vertex with some program behavior, E^+ is the transitive closure of the dataflow graph edge relation E, representing direct and indirect data dependencies, and E_r^+ is the transitive closure of the dependencies between record tuples as described previously in this chapter.

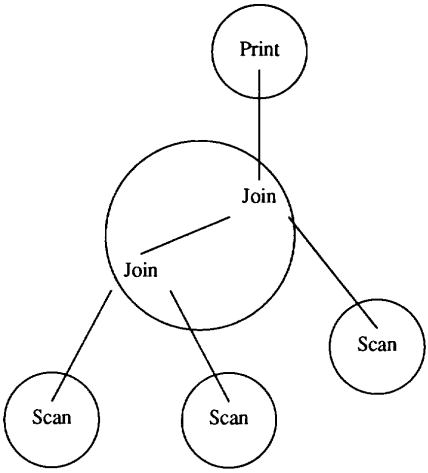


Figure 2.8: (Reproduced from [Gra94], page 131.) Vertical parallelism in Volcano.

2.4.2 Pipeline parallelism

Data transformations at operators that are, potentially indirectly, connected by a dataflow edge can still be executed in parallel in a streaming dataflow when the records being processed do not have a data dependency on each other. Intuitively, this is running in parallel separate tasks in a pipeline when they act on different data; for example an earlier vertex in the pipeline may process newer information while a later vertex processes information that reached the system earlier and has already progressed through the earlier vertex.

Formally, the code fragments $P(v_1)$ and $P(v_2)$ associated with two operators v_1 and v_2 for which $(v_1, v_2) \in E^+ \vee (v_2, v_1) \in E^+$ can be executed in parallel to generate record tuples r_1 and r_2 for which $(r_1, r_2) \notin E_r^+ \wedge (r_2, r_1) \notin E_r^+$. Figure 2.9 shows an example of pipeline parallelism available in a dataflow graph. The records r_{1x} and the records r_{2y} are unrelated in E_r^+, meaning there are no data dependencies between these two sets of records. For this reason, r_{24} and r_{13} can be generated in parallel at their respective operators; the same is true for r_{23} and r_{14}.

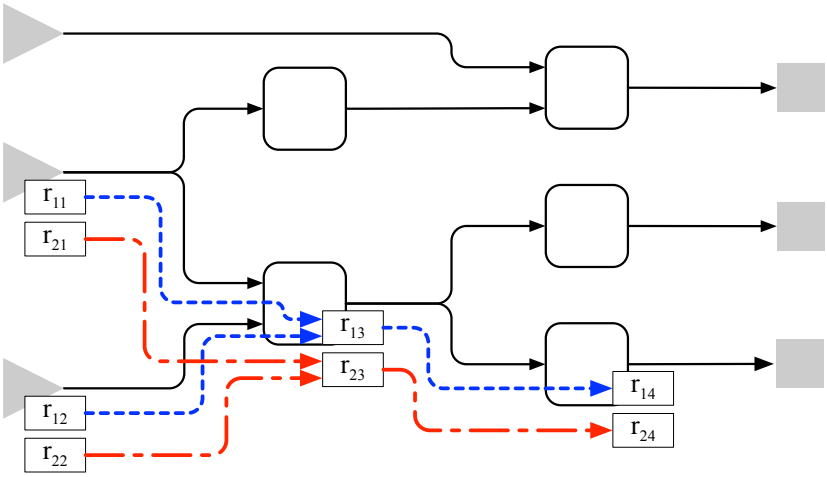


Figure 2.9: Example of pipeline parallelism for records in a dataflow graph.

Pipeline ('vertical') parallelism is described in Volcano as using each processor to run a subset of the operators in the dataflow graph, as depicted in Figure 2.8, where one processor runs each of the 'Scan' operations, one processor runs the two 'Join' operations, and one processor runs the 'Print' operation.

2.4.3 Intra-operator data dependencies

In many cases the generation of a record at a certain operator may also depend on a previous record having completed processing at the same operator. For example, a rolling aggregation may require all previous inputs having being completely processed before moving onto the next. This can be described by extending the relation E_r and its transitive closure E_r^+ with intra-operator data dependencies between records. Figure 2.10 extends the example of Figure 2.9 with intra-operator ordering dependencies; in this revised example r_{23} can be generated in parallel with r_{14} but r_{13} and r_{24} cannot because there is an indirect data dependency

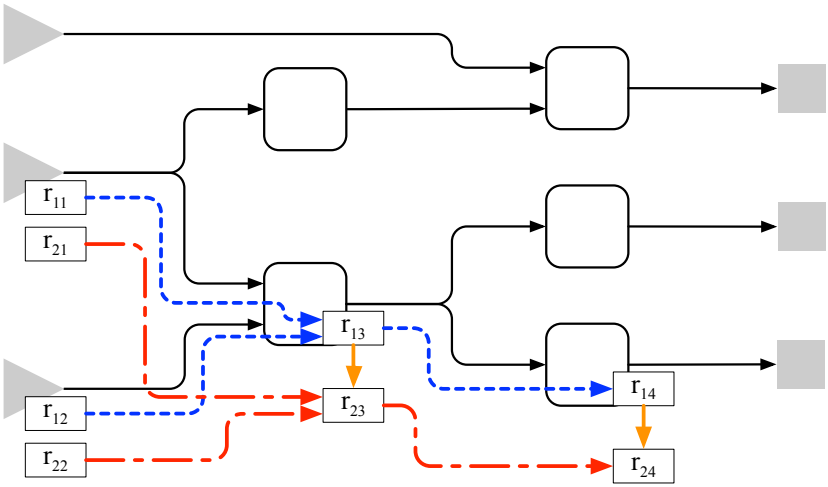


Figure 2.10: Example of intra-operator data dependencies for records in a dataflow graph.

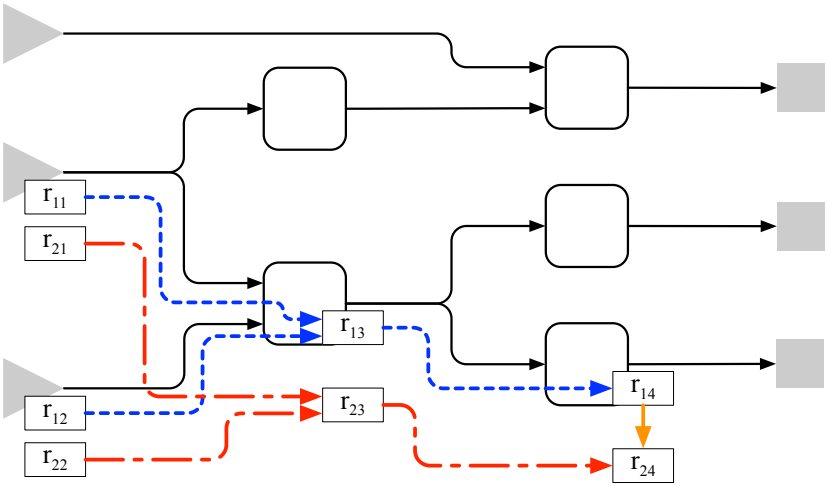


Figure 2.11: Example of data parallelism for records in a dataflow graph.

$(r_{13}, r_{24}) \in E_r^+$ due to $\{(r_{13}, r_{14}), (r_{14}, r_{24})\} \subseteq E_r$.

2.4.4 Data parallelism

When two (or more) records at the same operator do not have an intra-operator data dependency (in the extended E_r), the data transformation can be executed in parallel for both. This is typically data that relates to different entities, for example bids for different auctions.

Formally, the code fragment $P(v_x)$ associated with the operator v_x can be executed in parallel to generate record tuples r_1 and r_2 for which $(r_1, r_2) \notin E_r^+ \wedge (r_2, r_1) \notin E_r^+$. Figure 2.11 shows an example of data parallelism available in a dataflow graph. The records r_{13} and r_{23} are unrelated in E_r^+, meaning there are no data dependencies between them. For this reason, r_{13} and r_{23} can be generated in parallel; this is not true for r_{14} and r_{24} for which $(r_{14}, r_{24}) \in E_r^+$.

Volcano identified 'intra-operator parallelism' where multiple processors can be allocated to one (or more operators) to process subsets in a partition of the dataset (or intermediate result): Figure 2.12 shows a

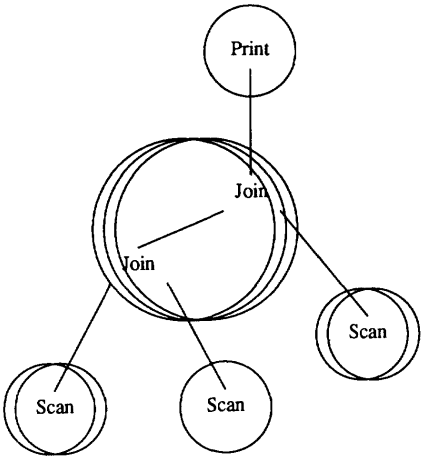


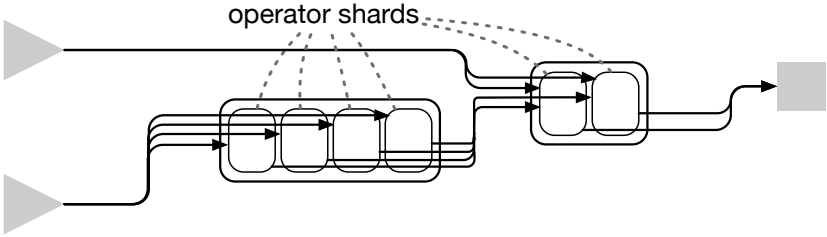
Figure 2.12: (Reproduced from [Gra94], page 132.) Intra-operator horizontal parallelism in Volcano.

Volcano execution plan where two of the 'Scan' tasks are handled jointly by two processors each, and the two 'Join' operations are handled by another two processors, each responsible of the data associated with a subset in a partition of join key.

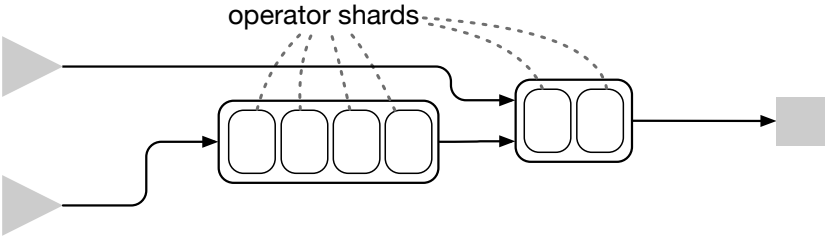
2.5 Operator shards

It is very often possible to group record tuples at an operator using a stream record key function $S : R \to K$ which selects a subset of the tuple attributes (e.g. Aurora's record key) in such a way that $\forall r_i, r_j : S(r_i) \neq S(r_j) \Rightarrow ((r_i, r_j) \notin E_r^+ \wedge (r_j, r_i) \notin E_r^+)$. In other words, the record tuples can be partitioned so that tuples in different subsets do not have a data dependency on each other: for example, bids can be grouped by auction at an operator that computes the top-k bids for each auction.

Any such partition of the record tuples at an operator yields subsets that can be processed in parallel. We can then re-interpret each operator as a collection of *operator shards*, one for each of these subsets. Most



(a)



(b)

Figure 2.13: Example of dataflow graph with sharded operators. This is a simpler dataflow graph than previous figures.

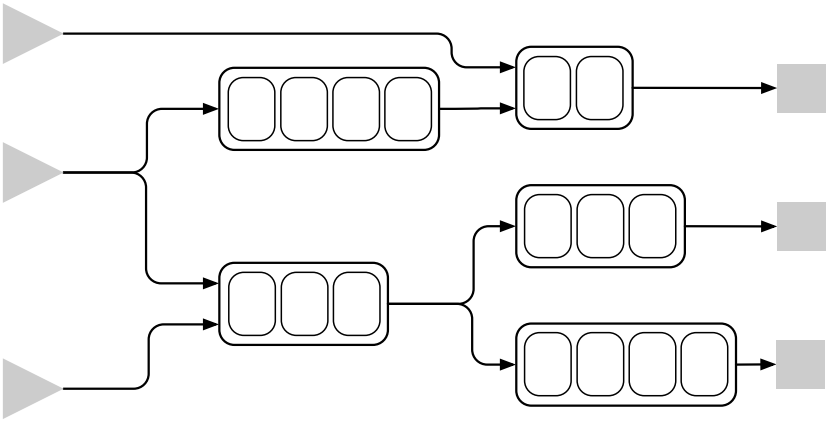


Figure 2.14: Example of data parallelism for records in a dataflow graph.

dataflow models and systems support sharding operators; the shards are sometimes known as *operator instances*.

Figure 2.13a depicts a dataflow where operators have been partitioned in different numbers of shards; if there was a dataflow edge between two operators in the original graph, there will be an edge for each pair of shards of the respective operators. This is just one example: for most dataflow programs and input datasets, there are many different ways to partition the record tuples.

We can simplify the graph again by having vertices (V) each represent a collection of shards, as depicted in Figure 2.13b; this is the model commonly used by popular dataflow models and systems. Figure 2.14 is the complete running example dataflow graph with all operators sharded.

2.6 Timestamps

In streaming dataflow programs that process incoming data continually (as opposed to a static input), like those used for monitoring services and on-line analytics, processing a new record may depend on a previous

record having completed processing, as described in subsection 2.4.3. This extends to output records, that (may) depend on a prefix of the input record history; to provide consistent results, it is necessary to encode such dependencies so that (i) only actural dependencies of an output record are used when computing it, and (ii) it is possible to determine when all the output records that depend on a certain input have been produced.

Timestamps are a widely-used to represent event ordering and causality [Lam78]. In the context of continuous queries, timestamps have been associated with input data records since the Tapestry system [Ter+92], are associated with all records in the system and used for QoS purposes in Aurora and Borealis [Aba+03; Aba+05], and STREAM [BW01; Ara+04] which defines "a stream S [as] an unbounded bag (multiset) of *pairs* $\langle s, \tau \rangle$, where s is a tuple and $\tau \in \Gamma$ is the timestamp that denotes the logical arrival time of tuple s on stream S" (reproduced from [Ara+04], page 2).

2.6.1 Encoding dependencies with timestamps

Following STREAM, in recent streaming dataflow systems **timestamps** are used to encode potential intra-operator dependencies and, in combination with the structure of the dataflow graph, potential dependencies on record tuples generated by earlier dataflow vertices (or in the inputs), as depicted in Figure 2.15. Intuitively, a record produced at a certain operator at a certain timestamp may depend on record tuples at an earlier vertex with the same or an earlier timestamp; it may also depend on record tuples at the same vertex but with an earlier timestamp.

Formally, in addition to previous relations and functions, we define:

- a set of timestamps T for which a transitive, reflexive, anti-symmetric partial order $\preceq$ is defined;
- the function $\mathcal{T} : R \to T$ which associates each record tuple with its timestamp;
- the function $\mathcal{V} : R \to V$ which relates a record to the vertex where it's generated (this formalizes the relationship between records and vertices discussed in subsection 2.3.2);

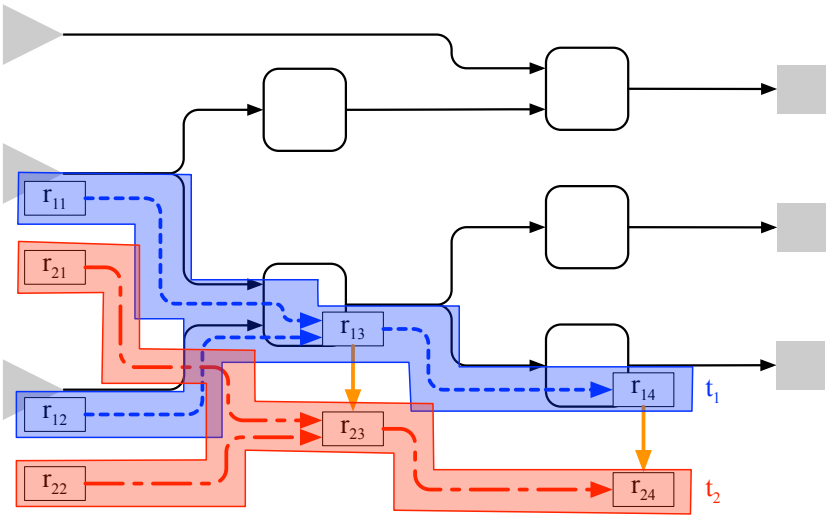


Figure 2.15: Example of timestamp encoding dependencies in a dataflow graph.

With these definitions, timestamps approximate the data dependencies E_r^+ as follows[2]:

$$(r_i, r_j) \in E_r^+ \rightarrow \mathcal{T}(r_i) \preceq \mathcal{T}(r_j) \wedge \Big(\mathcal{V}(r_i) = \mathcal{V}(r_j) \vee (\mathcal{V}(r_i), \mathcal{V}(r_j)) \in E^+\Big) \tag{2.1}$$

In other words, there may be a dependency between two records r_i and r_j if the timestamp of r_i is earlier than or equal to the timestamp of r_j, $\mathcal{T}(r_i) \preceq \mathcal{T}(r_j)$, and either (i) the records are generated by the data transformation at the same operator, $\mathcal{V}(r_i) = \mathcal{V}(r_j)$, or (ii) the records are generated at two operators that are connected in the dataflow graph, $(v_i, v_j) \in E^+$).

These timestamp-based dependencies define the set $\mathcal{E}_r^+$

$$\boxed{\mathcal{E}_r^+ = \Big\{(r_i, r_j) \Big| \mathcal{T}(r_i) \preceq \mathcal{T}(r_j) \wedge \Big(\mathcal{V}(r_i) = \mathcal{V}(r_j) \vee (\mathcal{V}(r_i), \mathcal{V}(r_j)) \in E^+\Big)\Big\}} \tag{2.2}$$

which approximate the exact data dependencies E_r^+:

$$\mathcal{E}_r^+ \subset E_r^+$$

2.6.2 Effect of timestamp approximation on available parallelism

As we have seen, a dependency $(r_i, r_j) \in \mathcal{E}_r^+$ indicates that there may be a data dependency on r_i when computing r_j, but that dependency may not actually exist. This cuts down on available parallelism in the model, because all

$$\Big\{r_k \Big| (r_k, r_x) \in \mathcal{E}_r^+\Big\}$$

would need to have been completed before a certain r_x can be generated.

Fortunately, the data transformation associated with a vertex v_x (via $P(v_x)$[3]) is sufficient information to determine the precise data dependencies (r_k, r_x) where $\mathcal{V}(r_x) = v_x$, i.e. all records generated at v_x.

[2] As a reminder, E_r represents record data dependencies, and E_r^+ is its transitive closure (subsection 2.3.2); E are the edges of the dataflow graph, and E^+ is its transitive closure (section 2.1).

[3] As a reminder: $P : V \rightarrow C$ associates each vertex with some program behavior.

For example, a simple data transformation for a vertex v_x that takes every integer input record r_x ($\mathcal{V}(r_x)$) and adds one to it (belonging to a class of transformations generally termed as `map`), encodes a direct dependency between each record generated r_x and the corresponding incoming record r_k at the immediately preceding operator ($(\mathcal{V}(r_k), \mathcal{V}(r_x)) \in E$) but no intra-operator dependency (r_p, r_q) between records at v_x ($\mathcal{V}(r_p) \neq v_x \wedge \mathcal{V}(r_q)$).

2.6.3 Towards time-aware dataflow

$\mathcal{E}_r^+$ loses precise dependency information when compared to the exact per-tuple data dependency information (lineage) in E_r^+ but enables tracking these dependencies by associating a timestamp to every record ($\mathcal{T}$) and using the structure of the dataflow graph ((V, E)).

The $\mathcal{E}_r^+$ approximation, in addition to vertex-local program behavior P that allows expressing finer grained dependencies, is the basis for the coordination mechanisms of recent time-aware dataflow streaming systems. We describe the time-aware dataflow model in the next chapter.

3 Time-aware dataflow

This chapter is based in part on "Verified Progress Tracking for Timely Dataflow" [Bru+21], *Timestamp tokens: a better coordination primitive for data-processing systems* [LM22], and "Shared Arrangements: practical inter-query sharing for streaming dataflows" [McS+20].

This chapter introduces the time-aware dataflow programming model. This model captures the core semantics of the underlying programming models of modern, record-at-a-time, low-latency, time-aware dataflow systems that define their semantics use timestamps to express which outputs are consistent in relation to which inputs: Apache Storm [Apa], Naiad [Mur+13; MT], timely dataflow [MT], Apache Flink [Car+15], and Google Cloud Dataflow [Aki+15].

These data-parallel stream processing systems express such computations as a dataflow graph whose vertices are *operators*, and whose roots constitute *inputs* to the dataflow. A *message* (*e.g.* a record representing an event in stream) arrives at an input with an associated timestamp and flows along the graph's edges into operators. Each operator takes the incoming message, processes it, and emits any resulting derived messages. Operators can also emit messages based on incoming control signals. Operators have access to *progress information* through control signals: they can determine which timestamps for messages can still reach their inputs because they are still being processed in upstream operators.

Chapter 2 described dataflow models in terms of the data dependencies and opportunities of parallelism they encoded. The programming model used by time-aware dataflow systems is designed to capture those

dependencies without the need to track per-record lineage graphs (like we did with E_r, E_r^+ in subsection 2.3.2 and section 2.4) and instead uses the dataflow graph, operator logic (program behavior of a vertex), and record timestamps for a conservative approximation of such dependencies (like we did with $\mathcal{E}_r^+$ in section 2.6).

In this chapter we refer to dataflow vertices V and their associated program behaviors P[1] as *operators*. We refer to record tuples as *messages* which flow along dataflow edges E which we call *channels*. This terminology is more common in recent time-aware dataflow streaming systems and is also used for our formal model and machine-checked proof in Chapter 8.

As we will see, *progress information* tracked by time-aware dataflow systems using control signals encodes these timestamp-based data dependencies to determine when all dependencies of a certain record r_x in $\mathcal{E}_r^+$ have been computed and delivered to the operator that generates it, $\mathcal{V}(r_x)$. Because $\mathcal{E}_r^+$ is a conservative approximaton of the exact dependencies, this is sufficient to determine that the data transfomration for r_x can proceed.

Some of the terminology used in this chapter follows that of *timely dataflow* [Mur+13; MT], in particular the concepts of operator "ports" and of "frontiers" as antichains.

3.1 Time-aware dataflow model

3.1.1 Dataflow graph

A time-aware dataflow computation is represented by a directed graph of operators (the graph's vertices), connected by channels (the graph's directed edges). Each operator in the dataflow graph is instantiated on one or more workers. Each instance of an operator is responsible for a subset, or shard, of the data being incoming to that operator: the system partitions the key space, and creates operators to independently process each partition. Operators communicate through channels which carry *messages*: these channels are reliable, asynchronous, and FIFO. Each message r_x is associated with a timestamp $\mathcal{T}(r_x)$. Figure 3.1 and

[1] As a reminder: $P : V \to C$ associates each vertex with some program behavior.

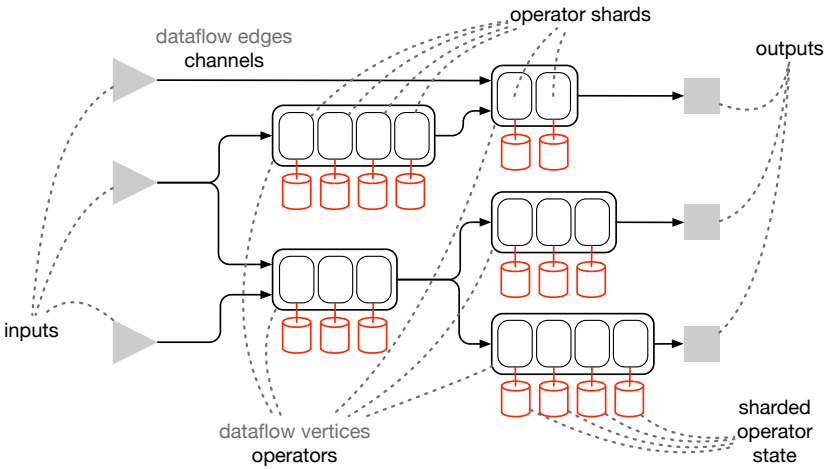


Figure 3.1: A time-aware dataflow graph.

Figure 3.2 show a time-aware dataflow graph, an operator, and the related concepts described in this section.

3.1.2 Operator state

In processing an update, a dataflow operator may refer to its *state*: long-lived information that the operator maintains across invocations. State allows for efficient incremental processing, such as keeping a running counter. For many operators, the state is indexed by a *key* contained in the input update. For example, a `count` operator over auction bids grouped by the user who posted them will access its state by user ID. Each operator instance has access to a shard of the operator state that corresponds to the key shard associated with that instance.

The operator state allows computing and storing partial results for data transformations where the resulting record r_x has data dependencies that span earlier timestamps: $(r_i, r_x) \in \mathcal{E}_r^+$ where $\mathcal{T}(r_i) \prec \mathcal{T}(r_x)$ and $\mathcal{T}(r_i) \neq \mathcal{T}(r_x)$.[2] An example of these dependencies is all the auction bids for the `count` operator described above: if the operator could not maintain state

[2] As a reminder: $\mathcal{T}$ associates each record to its timestamp, as discussed in section 2.6.

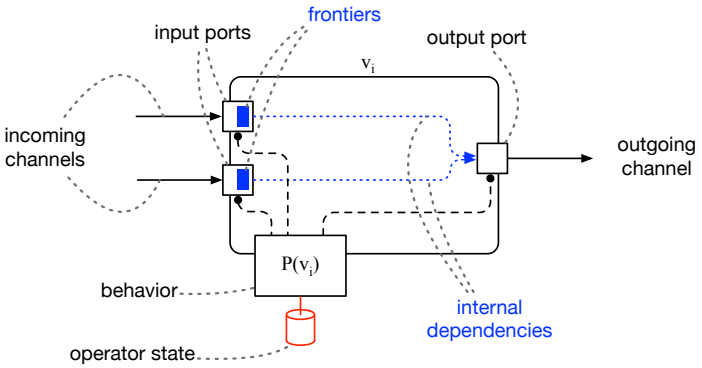


Figure 3.2: A time-aware dataflow operator.

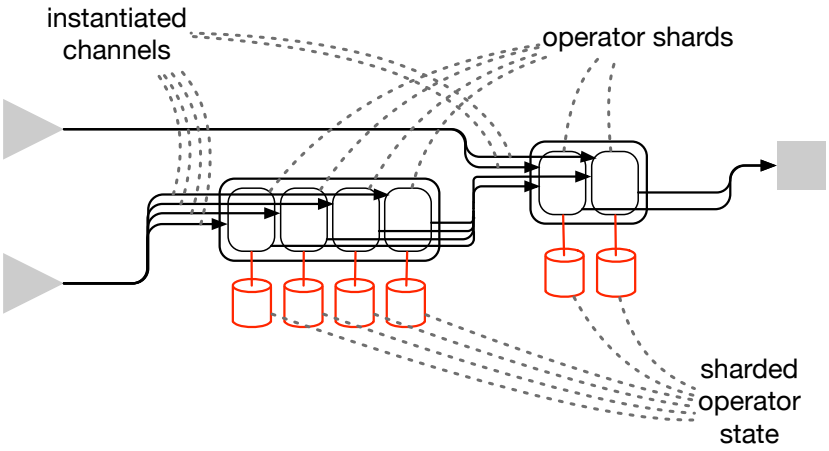
to summarize the input so far, it would need persistent access to all of its input messages for the entire duration of the computation.

3.1.3 Operator ports

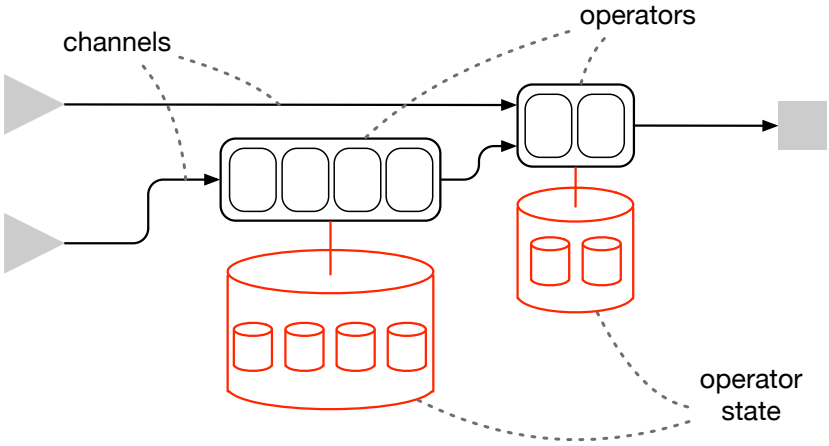
Each operator has an arbitrary number of input and output ports. An operator instance receives new data through its input ports performs processing, and produces data through its output ports. A dataflow channel is an edge from an output port of an operator o_a to an input port of an operator o_b and represents all FIFO channels between the outputs of all instances of o_a and the inputs of all instances of o_b.

3.1.4 Instantiated dataflow graph

Figure 3.3a depicts a time-aware dataflow graph where operators have been instantiated as multiple shards with their associated state shards; if there was a dataflow channel between two operators in the original graph, there will be a concrete cannel for each pair of shards of the respective operators.



(a)



(b)

Figure 3.3: Example of time-aware dataflow graph with sharded operators. This is a simpler dataflow graph than Figure 3.1

Similarly to section 2.5, we can simplify the graph again by having vertices (V) each represent a collection of operator shards and edges E represent a collection of edges in the instantiated dataflow graph, as depicted in Figure 3.3b; this is the model commonly used by popular dataflow models and systems.

In the following, when we talk about a dataflow operator, we refer to all its shards in the instantiated graph, and when we talk about a dataflow channel, we refer to all the associated concrete channels that connect operator instances.

3.1.5 Internal dependencies

Internal operator connections are edges from an input port to an output port which are described by **internal depency** information: which inputs and output ports are causally connected and what is the minimal increment to timestamps applied to data processed by the operator.

Most modern systems assume that all inputs are causally connected to all outputs and that the minimal increment to timestamps is zero, i.e. operators can produce output messages with the same timestamp as the input message that the output message depends on. The notable exception is the *timely dataflow* programming model [Mur+13], which allows operators to specify *summaries*: causal connections between the inputs and outputs of each operator and the minimum timestamp increment on that connection. This minimum increment information is necessary for *timely dataflow* to support cycles in the dataflow graph without livelocks.

3.1.6 Frontiers

Operator instances must be informed of which timestamps they may still receive from their incoming channels, to determine when they have a complete view of data associated with a certain timestamp. The system's coordination mechanism tracks relevant *progress information* and summarizes it to one frontier per operator input port. A frontier is a lower bound on the timestamps that may appear at the operator instance inputs. It is represented by an antichain[3] F indicating that the operator may still receive any timestamp t for which $\exists t' \in F.\ t' \preceq t$.

[3] "An antichain is a subset of a partially ordered set such that any two distinct elements in the subset are incomparable." (from "Antichain" on Wikipedia [Wik])

3.2 Encoding data dependencies

Timestamps and frontiers encode data dependencies. The time-aware dataflow system coordination mechanism is tasked with computing frontiers based on *progress information* to ensure results are correctly computed. Frontiers properly respect data dependencies when the following property holds.

$$\forall v \in V, r_v \Big| \mathcal{V}(r_v) = v \rightarrow \Big((\exists r_a | (r_a, r_v) \in \mathcal{E}_r^+ \ \wedge \neg completed_v(r_a)) \rightarrow frontier_{\mathcal{V}(r_a)}(v) \preceq \mathcal{T}(r_a) \Big) \tag{3.1}$$

where

- $completed_v(r)$ indicates that all records r_i (messages) that depend on r and for which $(\mathcal{V}(r_i), v) \in E$ have been delivered to v;
- $frontier_{v_p}(v)$ is the frontier on the input(s) of operator v reachable from v_p;
- we use $F \preceq t$ as a shorthand to indicate that $\exists t_f \in F | t_f \preceq t$.

That is, the frontier at an operator's input port must contain one element t_f that is less or equal $\preceq$ to (i) every timestamp that has not been retired at any upstream operator (from which the input port is reachable), and (ii) to every timestamp of messages in-flight on upstream dataflow edges (from which the input port is reachable). A timestamp has been retired on an output of a certain operator when the operator has produced all messages with that timestamp on that port.

Frontiers for each input port in a time-aware dataflow graph move forwards through time (as defined by $\preceq$) as the computation advances and timestamps are retired at operators and messages are delivered.

The concept of approximating data dependencies with using auxilliary information is common in Computer Science and appears in stream processing as "punctuated streams" [Tuc+03]: "punctuation" indicates that all data pertaining to a certain logical substream has been delivered, similarly to how a frontier advancing past a timestamp indicates that all input data pertaining to that timestamp has been received.

Monotonic frontiers Frontiers, like timestamps, are a conservative approximation of the exact data dependencies E_r^+. Because the operator's behavior $P(v_i)$ only has access to the frontier to determine when it can proceed to compute results for a certain timestamp, it is important that the frontier at each operator input port advances monotonically during the computation and never moves back in time (i.e. elements of a frontier should each be beyond the prior frontier).

3.2.1 Coordination with timestamps and frontiers

In a pipelined, data-parallel time-aware dataflow system, concurrent updates may race along the multiple paths (and even cycles) between dataflow operators potentially distributed across multiple threads of control, and arrive in different orders than they were produced. With logical timestamps on messages and timestamp frontiers from the system, operators can maintain clear semantics even with asynchronous, nondeterministic execution.

A system should guarantee that all future timestamps received at an operator input are beyond the frontier most recently reported by the system, and that these reports should only advance.

3.3 Time-aware dataflow systems

This model extends and abstract the timely dataflow programming model introduced in Naiad with the goal to capture the programming model of other modern low-latency time-aware dataflow systems. In this section, we descrive representative systems; subsection 3.3.3 discusses timely dataflow.

3.3.1 Spark Streaming

Spark [Zah+12] is a batch system that models a computation as an acyclic dataflow graph, but without distinct logical times: inputs in Spark are either "complete" or "not yet complete". The Spark system tracks which inputs are complete and signals operators when their inputs are all complete and the operator can run to completion. Operators report back to the system as they complete their outputs.

Spark Streaming [Zah+13] adapts Spark to a streaming setting. Spart Streaming partitions logical time into small batches, and for each batch evaluates an entire dataflow. It therefore implicitly provides timestamps, with progress indicated by the scheduling of an operator. Spark Streaming operators do not have long-lived state, but each invocation can read an input corresponding to its prior state and write an output for its updated state, at greater expense than updating in-memory state.

3.3.2 Flink

Flink [Car+15] models computations as an acyclic dataflow graph, with integer logical times. Flink streams (dataflow edges) report an increasing integer "watermark" lower-bounding the timestamps the stream may yet produce. These watermarks are interleaved in the stream of data itself, and each operator is required to produce them in their output streams as well. Flink does not have a centralized scheduler, and maintains a fresh view of its outputs only through the continued introduction of new watermarks in the dataflow inputs. In Flink a "watermark" for a timestamp t indicates that all messages that follow have timestamps greater or equal to t. Flink uses "watermarks" to maintain frontier information for the operator inputs. Flink operators can have long-lived state, and can themselves be the result of sharding a larger dataflow operator.

3.3.3 Timely dataflow

Timely dataflow is a model for data-parallel dataflow execution, introduced by Naiad. Timely dataflow models computations as a potentially cyclic dataflow graph, with partially ordered logical times. Each timely dataflow operator is sharded across all workers, with data exchanged between workers for dataflow edges where the destination operator requires it. In timely dataflow, all data carries a logical timestamp, and workers exchange timestamp progress statements out-of-band. Workers independently determine frontiers for each of their hosted operators.

Naiad operators request "notifications" at specified logical times, and Naiad invokes a callback only once it determines that all messages bearing that logical time have been delivered. Naiad does not present operators with lower bounds for their inputs, and instead requires operators to defer the responsibility of scheduling to the system itself, in part because the

logic for doing so requires a holistic view of the dataflow graph and all other pending notifications.

Timely dataflow [MT] is also the name of the current implementation of timely dataflow which differs from the original model in two ways relevant for the work in this thesis.

Timestamp partial order In Naiad, timestamps are tuples of integers $t = (e, c_1, c_2, ...)$ where the "epochs" e are externally assigned and the iteration counters c_1, c_2, etc. are used to track the number of times a piece of data has gone through a feedback edge in a cycle in the dataflow graphs. There is a total order on epochs, so timestamps with a lower epoch number are $\preceq$ of timestamps with an higher epoch number. Within the same epoch, timestamps are partially ordered. In modern timely dataflow, timestamps can be of an arbitrary type for which a partial order $\preceq$ is defined; the generalization to arbitrary, partially ordered ($\preceq$) timestamps precedes the work in this thesis.

Frontiers In modern timely dataflow, the system computes frontiers for operator inputs using a distributed protocol; however, before the work in this thesis, frontiers were not accessible by the operator behavior code and operators were required to request "notifications" at specified timestamps, like in Naiad. Chapter 4 describes a new operator interface that we designed to allow operators to directly access and manipulate frontiers.

4 Timestamp tokens

This chapter is based on *Timestamp tokens: a better coordination primitive for data-processing systems* [LM22]; some of the work presented in this chapter was initially reported as part of my Master Thesis "Programmable scheduling in a stream processing system" [Lat16].

As we have seen in Chapter 2 systems for data-intensive computation have advanced through programming models that allow programs to reveal progressively more opportunities for concurrency. Frameworks like MPI [MPI] allow programmers only to explicitly sequence data-parallel computations. Systems like DryadLINQ [Yu+08] and Spark [Zah+12] use data-dependence graphs to allow programs to express task parallelism. Stream processors like Flink [Car+15] and Naiad [Mur+13] (following [Apa; Aki+13; CGM09]) add a temporal dataflow dimension to represent pipeline parallelism. In each case, new runtimes extract more detailed information about the computations, allowing them greater flexibility in their execution.

This chapter proposes a new programming abstraction for time-aware dataflow systems (as described in Chapter 3): a coordination primitive that dataflow operators use to explicitly signal which timestamps are in-progress or retired. Drawing inspiration from work on capability systems, this new coordination primitive, the *timestamp token*, is an in-memory object that can be held by an operator and provides the ability to produce timestamped data messages on a specific dataflow edge.

A timestamp token does not require repeated interaction between system and operator to confirm, exercise, or release this ability. Instead, an operator accumulates and summarizes its interactions with its timestamp

tokens. The system collects this information when most convenient, maintains a view of outstanding timestamp tokens, and provides summaries of potential input timestamps to each operator.

As we will see in the following chapters, this abstraction is sufficient to construct complex synchronization protocols like the ones necessary for system mechanisms such as index sharing, data re-partitioning, and fault tolerance without modifying the core system. These idioms could not be expressed easily, if at all, on top of other existing platforms. The approach presented here benefits from the ability to use timestamp to both (i) determine which timestamps can be retired at which operator and (ii) build the synchronization protocols for those system mechanisms.

My early prototype for a dataflow flow control mechanism, Faucet [LMC16; Lat16], uses timestamp tokens to allow operators (and dataflow fragments) to implement their own flow control, without modifying system code. DD, presented in Chapter 6, uses timestamp tokens to provide arbitrary granularity timestamps for differential dataflow [McS+13], dramatically improving the throughput over the corresponding Naiad implementation. Megaphone [Hof+19] uses timestamp tokens to specialize the implementations of operator-internal schedulers, for example using priority queues in operators that support them without requiring system-wide support. In each case, timestamp tokens' separation between system and operators provided the flexibility to introduce behavior that would otherwise require the implementation of a specialized system.

4.1 Background

Dataflow systems have become limited by the complexity of the boundary between system and computation. Specifically, as computations provide progressively more fine-grained and detailed information about concurrency opportunities, the scalability and sophistication of the system schedulers must increase. In our experience, system complexity has increased to the point that scheduling rather than computation becomes the bottleneck that prevents higher throughputs and lower latencies.

System designers have the opportunity to reduce the *volume* of coordination by reconsidering the interface between system and operator. For example, where Spark Streaming [Zah+13] must schedule distinct events to implement distinct logical times, Flink (and other stream processors)

allow operators to retire batches of events corresponding to blocks of logical times, substantially improving throughput.

Flink (and other stream processors) requires continual interaction with operators to confirm that they have no output at a logical time: operators are presented with periodic updates on the state of the system, act on them sequentially, and are tasked with propagating the coordination signal. With this interface the system must periodically poll all operators to make progress, even when no work needs to be done, and must forward all periodic updates, even when no operator has interested in them.

Naiad avoids periodic polling by asking operators to explicitly identify future times at which the operator should be notified, but requires the operators to register a notification handler that's driven by the system scheduler. Because the system sequences the operator's work by invoking the handler, the operator is not free to intelligently reorder work for increased efficiency.[1]

These interfaces reduce the volume of coordination, but require a deeper involvement of the system itself: continually invoking operators in Flink and sequencing notifications in Naiad. Each of these systems introduce new opportunities for concurrency, and corresponding performance gains on important tasks. However, no one system unifies the work of the others.

We believe that unifying this work, and laying the groundwork for more advanced behaviors, requires a *simplification* of the interface between system and operator, rather than further sophistication. The programming model proposed in this chapter exploits the unified model presented in Chapter 3.

4.2 Timestamp tokens

We propose that dataflow systems and operator logic can coordinate precisely, efficiently, and ergonomically by explicitly handling in-memory tokens that represent their ability to produce outgoing data in the future. We borrow and adapt this idiom from capability systems (e.g. object-

[1] The implementation of timely dataflow, before the work in this dissertation, presented a raw interface to the operators that enabled efficient processing but – in our experience – was very prone to subtle programming errors that caused hard-to-debug concurrency bugs.

capability systems [DV66; Fab74], capability-based protection and security [CJ75; Mul+90], hardware capabilities [A F73; Chi+15]). Similarly to capabilities[2], a timestamp token represents a computing object – an operator output – and the actions that can be performed with respect to that object: the production of data at timestamp t and dataflow location l.

Following Naiad we refer to the pair of timestamp t and location l as a *pointstamp* (t, l). A location can be either a node in V or an edge in E.

Definition. A *timestamp token* is a coordination primitive that names an associated pointstamp (t, l), and which gives its holder the ability to produce messages with timestamp t at location l.

The location for a timestamp token is one of the output edges of the operator that holds it.

Notwithstanding any other similarities to capabilities, our interest is in the information that holding timestamp tokens communicates to others. The system tracks the set of live timestamp tokens and summarizes this information to operators as *frontiers*: lower bounds on the timestamps that operators may yet observe in their inputs. By downgrading (to future timestamps) or discarding their held timestamp tokens, operators allow frontiers to advance and the computation as a whole to make forward progress.

4.2.1 The timestamp token life-cycle

Each dataflow operator is initially provided with a timestamp token for each of its output edges, each bearing some minimal “zero" timestamp. This gives each operator the opportunity to be a source of timestamped messages, even without receiving input messages. For many operators, their first actions will be to discard these timestamp tokens, by which they release their ability to produce output messages unprompted, and unblock the dataflow system at the same time.

As a dataflow operator executes, it can receive, exercise, downgrade, and discard timestamp tokens as depicted in Figure 4.1.

[2] “Each capability [...] locates by means of a pointer some computing object, and indicates the actions that the computation may perform with respect to that object.” (reproduced from [DV66], page 145)

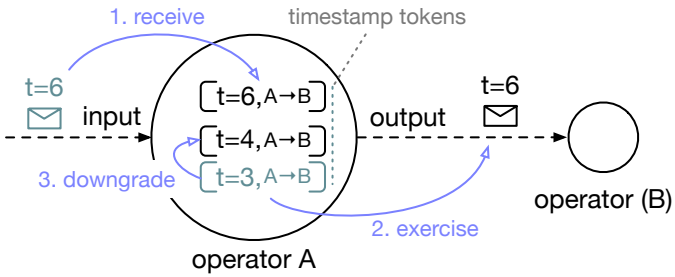


Figure 4.1: Timestamp token life-cycle.

1. **receive**. Operators invoke a *receive* operation to receive timestamped input messages, each of which provides a timestamp token at that timestamp for each of the operator's outputs.

2. **exercise**. Operators can produce timestamped output messages as long as they hold a timestamp token with the corresponding timestamp and output edge.

3. **downgrade and discard** Lastly, operators can arbitrarily hold, downgrade (to future timestamps), and discard their timestamp tokens as their logic dictates.

The dataflow system is informed of the net changes to the number of timestamp tokens for each pointstamp, but only passively in response to operator actions, rather than actively as a gatekeeper. Through this information the system can inform dataflow operators about the consequences of operator actions, without the specific details of the reasons for those actions.

4.2.2 Coordination

The coordination state of the dataflow system is the set of timestamp tokens, which when combined with the dataflow graph determines lower bounds for the timestamps at each operator input. As the set of timestamp tokens evolves these lower bounds advance, and the dataflow system

has the responsibility of informing operators as this happens. The difference with timestamp tokens is that operators drive the *production* of this information, instead of the system itself.[3]

Operators have a great deal of flexibility in how (or even if) they respond to changes in their input frontiers (timestamp lower bounds). Certain streaming operators like `map` and `filter` can be oblivious to this information and process data as it arrives. Synchronous reduction operators like `reduce` should await the indication that they have received all inputs for a timestamp before they apply their reduction function and produce output. Hybrid operators like `count` may perform some accumulation in place and await the frontier advancing before producing the final tally for each timestamp. In each case the operator responds to input data and changes in its input frontiers, with output data and changes in its held timestamp tokens, but does not otherwise expose complexity to the system.

4.3 Implementation

We implemented timestamp tokens for Timely dataflow [MT] in the Rust programming language [Rus] [MK14]. In our implementation, timestamp tokens are Rust types that wrap a timestamp, a location, and a bookkeeping data structure shared with the system. Operator logic manipulates timestamp tokens through their methods—cloning, downgrading, and discarding them—which update the shared data structure with integer *changes* to the numbers of timestamp tokens at each timestamp and location.

The timely dataflow system drains shared bookkeeping data structures outside of operator logic but on the same thread of control, which ensures the changes reflect atomic operator actions. Following Naiad's progress tracking protocol, these collected changes are broadcast among un-synchronized workers. Any subset of atomic updates forms a conservative view of the coordination state (the outstanding timestamp to-

[3] For example, Naiad does not allow operators to hold tokens across invocations; Timely Dataflow (without timestamp tokens) does, by allowing operators to participate directly (and often incorrectly) in the coordination protocol. Here, timestamp tokens are respectively more expressive, and safer.

kens) and is sufficient to maintain a conservative view of timestamp lower bounds for each operator across the otherwise asynchronous workers.

The Rust [Wei19] language provides several features that simplify our implementation. Rust is type-safe, and users cannot fabricate timestamp tokens outside of unsafe code. Rust also does not allow users to destructure private `struct` fields, ensuring that we protect the shared bookkeeping data structure from direct user manipulation. Rust's affine type system ensures that users cannot casually copy timestamp tokens without explicit method calls, which allow us to interpose and increment counts. Finally, Rust eagerly invokes destructor logic, so that discarding a timestamp token is immediately visible to the system.

We have implemented Naiad notifications in library operator logic, and if in each invocation an operator processes only their least timestamp they reproduce Naiad's notification behavior. We can also implement Flink-style watermarks, with operators that explicitly hold timestamp tokens for their output watermarks and downgrade them whenever these watermarks advance. Both these idioms are helpful but restrictive, and they are enforced system-wide in prior work. Our intent is that operators should be able to choose the most appealing idiom, or new idioms as appropriate, without requiring the system to change as well.

This generality is not without some ergonomic cost: prior systems could more easily encourage operators make forward progress. Flink operators should eventually bring their output watermarks in line with their input watermarks, and Naiad operators should respond to notifications with something other than a re-notification request for the same time. From experience, user operators can more easily "lose track" of a timestamp token, for example when used as a key in a hash map and not discarded once its associated values have been processed. We use Rust's type system to raise the programmers awareness, by providing operators only a "timestamp token option", which the operator must then specifically `retain` to receive a timestamp token. Rust's lifetime system ensures at compile time that the options themselves can not be held by an operator, forcing it to explicitly `retain` or pass on timestamp token options.

4.3.1 Timestamp tokens in code

We present an extract of the main definitions of the timestamp token Rust API and implementation in Figure 4.2.

```
/// The ability to send data with a
/// certain timestamp on a dataflow edge.
pub struct TimestampToken<T: Timestamp>Ⓐ {
  timeⒷ: T,
  bookkeepingⒸ: Bookkeeping<T>,
}

impl<T: Timestamp> TimestampToken<T> {
  /// The timestamp associated with this
  /// timestamp token.
  pub fn timeⒹ(&self) -> &T { ... }

  /// Downgrades the timestamp token to
  /// one corresponding to 'new_time'.
  pub fn downgradeⒺ(
    &mut self, new_time: &T) { ... }
   ...
}

impl<T: Timestamp> CloneⒻ
  for TimestampToken<T> {
  fn clone(&self) -> TimestampToken<T> { ... }
}

impl<T: Timestamp> DropⒼ
   for TimestampToken<T> {
  fn drop(&mut self) { ... }
}
   ...
```

```
impl<T: Timestamp, ...> OutputHandle<T, ...>Ⓗ
{
  /// Obtains a session that can send data
  /// at the timestamp associated with
  /// timestamp token 'tok'.
  pub fn sessionⒾ(
    &mut self, tok: &TimestampToken) -> Session<T, ...>
   { ... }
}
```

Figure 4.2: An extract of the timestamp token API and implementation in timely dataflow. We use circled letters, similar to Ⓩ, to mark points of interest in the code.

A `TimestampToken`(A) wraps a timestamp(B) and a bookkeeping data structure(C) shared with the system. These fields are private and the operator code cannot directly access or mutate them. The bookkeeping data structure records the location for which the `TimestampToken` is valid, which will be checked by the system should the `TimestampToken` be exercised to send data. Operators may hold any number of `TimestampToken`s.

Three methods, `downgrade`(E), `clone`(F), and `drop`(G), are the only ways user code can directly manipulate the number of timestamp tokens at a pointstamp (without the use of Rust's `unsafe` keyword). The number of timestamp tokens at a pointstamp is indirectly manipulated by sending timestamped messages to the location of that pointstamp, through the `session`(I) method.

Operator code can directly downgrade a timestamp token to a later timestamp with `downgrade`. This reduces the operator's ability to produce output at the wrapped timestamp, potentially to the point that the system can unblock downstream operators, though not beyond the timestamp downgraded *to*. The implementation of `downgrade` updates the bookkeeping data-structure to inform the system of the net changes to the number of timestamp tokens for each pointstamp.

Operator code can also call into Rust's `clone` (deep copy) and `drop` (destructor) methods on `TimestampToken`. Custom implementations of these two methods respectively increment and decrement pointstamp counts for the wrapped timestamp in the bookkeeping data-structure. A `drop` call is automatically inserted by the Rust compiler whenever an object goes out of scope, and makes it much less likely that an operator will fail to release a timestamp token.

In order to transmit data along an output dataflow edge, an operator must *express* a timestamp token. Access to outputs is guarded by an `OutputHandle`(H), whose method `session`(I) will create an active `Session` only when presented a reference to a `TimestampToken`. The `Session` is only valid for the wrapped timestamp, or timestamps greater than it. Rust's lifetime system ensures that the `TimestampToken` cannot be modified or dropped as long as the `Session` is active (until it is dropped). As long as the `Session` is available to user code, the `TimestampToken` is guaranteed to still exist unmodified. Sent data arrive at the destination bearing a timestamp token that can be used by the recipient.

4.3.2 Ergonomic modifications

The core timestamp token code is explained in the previous section, but we have also made several ergonomic improvements in at attempt to minimize the chance of unintended errors.

In addition to `TimestampToken` objects, which are *owned* by the code and data structures that reference them, we also provide a `TimestampTokenRef` structure that cannot be held longer than a fairly narrow lexical scope. To acquire an owned token, user code must explicitly call `retain` which then results in a `TimestampToken`. We have found this reduces the incidences of user code unintentionally capturing and indefinitely holding a timestamp token, thereby stalling out dataflows.

Both `TimestampToken` and `TimestampTokenRef` implement a Rust trait `TimestampTokenTrait` that allows system code (specifically `session`) to accept either. This allows users to bypass the `retain` method and create a `Session` from a token reference, avoiding some syntax but importantly also avoiding bookkeeping when timestamp token ownership is not needed.

Timestamp tokens by default update shared bookkeeping data structures, but do not force the system to immediately act upon the changes they reflect. The operators that house an `OutputHandle` inform the system that it should consult the shared bookkeeping, when the operator yields control. Several variants of `TimestampToken` take specific action when modified, including notifying the system that it should accept any updates and act on them. This allows these timestamp tokens to be used outside of the operators their pointstamps reference, and are especially useful for manual control of inputs to a dataflow when the logic cannot easily be encapsulated in an operator.

These modifications do not change the core behavior of timestamp tokens, but instead demonstrate how rough edges can be sanded down using layers atop timestamp tokens.

4.4 Example

We use the example of *tumbling windowed average* to demonstrate the lifecycle of timestamp tokens and how it generates coordination information. This operator receives timestamped integer-valued messages and reports

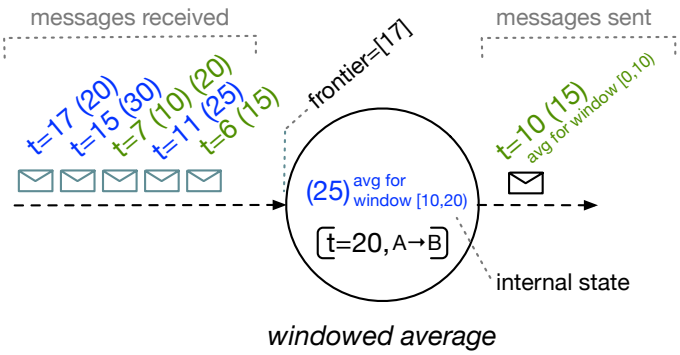


Figure 4.3: The *windowed average* operator in a sample execution with its internal state, a held timestamp token, and the data sent so far on its input and output edges.

the average every 10 timestamp units, at the timestamp of the start of the next window. The operator produces no output for windows which contain no data. Figure 4.4 list the example code.

Importantly, this is code that one can write to introduce the behavior of a tumbling window to a system. It is not code that an end user should be expected to write each time they want a tumbling window. Rather, it can be written once, and then end users can simply invoke the method with appropriate parameters.

Figure 4.3 is a snapshot of the execution after the output for the window $[0, 10)$ has been produced. At this stage the operator maintains the current average for open windows (for which some data has been received but not necessarily all data) and a timestamp token to produce the output at the timestamp of the next open window (in the Figure, time `20`).

The operator has great flexibility in how it implements its specification. For example, the operator can choose to retain only the timestamp tokens for timestamps that are not greater than some other held timestamp token, reducing system interaction at the cost of local bookkeeping. The operator can use ordered data structures to efficiently retire multiple windows at once, should the frontier advance suddenly. The operator can maintain partial aggregations for out-of-order data while still being clear at which times they might emerge.

We walk through the sample code in subsection 4.4.1 and call out the benefits timestamp tokens provide in subsection 4.4.2

4.4.1 Example code

Figure 4.4 shows the code listing for one of the many possible implementation of the *tumbling windowed average* operator described in section 4.4. The code presented closely resembles the real implementation of the operator, with some minor syntax modifications to aid readability and avoid Rust-isms that can be unfamiliar to the reader. Although detailed, this is the implementation expected of the system implementor; we expect end users would then access this functionality through a layer of abstraction rather than write it themselves.

The outer anonymous function (B) is invoked once by the system to intitialize the operator with a default timestamp token (C) at time `0` (D), which is immediately dropped (E). The operator initializes an ordered map (F) to store partial state for open windows: the timestamp for the end of the window maps to a tuple carrying the corresponding timestamp token and the partial `WindowData` (A) (the partial sum and count).

The inner anonymous function (G) contains the operator logic that is invoked every time the operator is scheduled. For each batch of input messages at a certain timestamp (I), it computes the end-of-window timestamp (J) from the message timestamp wrapped in the timestamp token `tok_ref` (H) (in the form of a `TimestampTokenRef`). If it has not seen data for this window before (K), it captures (L) the timestamp token, immediately downgrades it to the end-of-window timestamp, and stores it along with initialized empty window data into the `windows` map.

The timestamp tokens stored in the map implicitly inform the coordination state of the operator: the system is informed of pointstamp changes after each invocation of the operator logic caused by `retain`, `downgrade`, and `drop` (when a timestamp token is finally removed from the map and dropped).

For each batch of input messages the operator logic obtains a mutable reference (M) to the corresponding window data in the map, and updates the partial sum and count with each data point. Processing of new input concludes here.

The operator logic then needs to determine which windows have closed and emit the computed averages for them. This information is based on

```
1  /// User-defined structure to maintain window data.
2  struct WindowData(A) { pub sum: u64, pub count: u64 }
3  pub fn singleton_frontier(frontier: &MutableAntichain<u64>) -> u64 {
4    frontier.frontier().first().cloned().unwrap_or(u64::MAX)
5  }
     ...
6  // The `unary_frontier` method defines a new operator from a anonymous
       function that specifies its logic.
7  stream.unary_frontier(
8   Exchange::new(|x| x % (peers as u64)), "tumbling window", (B)|(C)tok,
      _info| {
9    (D)assert!(*tok.time() == 0);
10   (E)std::mem::drop(tok);
11   let mut windows: BTreeMap<u64, (TimestampToken<u64>, WindowData)>(F) =
       BTreeMap::new();
12   // Define the anonymous function that is repeatedly invoked with input
       and output handles.
13   (G)move |input, output| {
14     for (tok_ref(H), batch) in input {(I)
15       (J)let window_ts = round_up_to_multiple(*tok_ref.time(),
       WINDOW_SIZE);
16       if !windows.contains_key(&window_ts) {(K)
17         let mut window_tok = (L)tok_ref.retain();
18         window_tok.downgrade(&window_ts);
19         windows.insert(window_ts, (window_tok, WindowData { sum: 0, count
       : 0 }));
20       }
21       let (_, ref mut window_data(M)) = windows.get_mut(&window_ts).
       unwrap();
22       for d in batch {
23         window_data.sum += *d; window_data.count += 1;
24       }
25     }
26     let target_ts = singleton_frontier(input.frontier()(N));
27     for (_, (tok(O), window)) in windows.range(0..target_ts) {(P)
28       (Q)output.session(&tok(R)).give(window.sum as f64 / window.count as
        f64);
29     }
30     (S)windows.remove_range(0..target_ts);
31   }
32 })
```

Figure 4.4: A possible implementation of the tumbling window average operator described in section 4.4. We use circled letters, similar to (Z), to mark points of interest in the code.

the set of live timestamp tokens in the system and is summarized by the system as per-input *frontiers* at each operator: `input.frontier()` Ⓝ. In general, timestamps in timely dataflow can be multidimensional and result in frontiers defined by multiple minima, but in this case we know that timestamps, and consequently frontiers, are represented by a single unsigned integer value. The frontier value represents the lower bound on timestamps that may still appear on the input: consequently we can safely retire all windows with end-of-window timestamps up to, but excluding, the frontier (`target_ns`).

We leverage the map order to iterate over all open windows up to `target_ns` Ⓟ, and because we stored timestamp tokens alongside the window data, we obtain them during iteration Ⓞ and can immidiately leverage them to emit the computed averages at the correct timestamps Ⓠ: to do so, we are required to pass in a reference to the timestamp token Ⓡ. This ensures at compile time that the operator logic has the capability to send data at a certain timestamp.

The operator logic finally drops from the map all the windows it has just processed Ⓢ. The `drop` code for the timestamp tokens stored in the values removed from the map are invoked automatically (and eagerly): this again updates the pointstamp changes that are reported to the system, and ensures that the frontiers for other downstream operators are updated accordingly.

4.4.2 Benefits

The operator implementation above has several benefits that are prevented in other systems.

In a Spark-like system, where an operator is scheduled for each distinct timestamp, the operator would be unable to retire blocks of times concurrently. This limitation harms the throughput of data loading, and lowers the operator's throughput when bursts of differently timestamped data arrive. With timestamp tokens entire intervals of time can be closed at once, and the operator can perform all consequent work concurrently.

In a Flink-like system, the operator must be continually interrogated to advance its output watermark. Even if the operator input is idle for periods of time, the operator must remain active to inform downstream operators that there is no data. This scenario is more common than it might seem, with monitoring applications like fraud detection in which

one wants to quickly confirm the absence of results. With timestamp tokens the system can bypass the operator entirely, reducing compute load and the critical path latency.

In a Naiad-like system, the operator must defer scheduling to the system. Should a batch of times be retired at once, as when a watermark finally arrives, the operator must repeatedly yield to the system and be re-invoked with advancing timestamps. With timestamp tokens the operator can perform this work on its own, using an efficient ordered data structure.

In addition, timestamp tokens avoid restrictions on dataflow structure, for example the requirement (seen in Spark and Flink) that dataflow graphs be acyclic. Each of these benefits derive from involving the system *less*, instead providing the operator with both more information and more agency.

4.5 Evaluation

Our hypothesis is that by reducing systems complexity and granting more control on scheduling to individual operators, timestamp tokens remove the scheduling bottleneck that prevents modern data processing systems from reaching higher throughputs and lower latencies. We evaluate this hypothesis with a set of microbenchmarks designed to compare the different coordination mechanisms in prior art with timestamp tokens (subsection 4.5.2 and subsection 4.5.3) and with more complex workloads that attempt to replicate real-world operating conditions (subsection 4.5.4). We hope to observe that timestamp tokens operate robustly in all settings where any coordination mechanism avoids collapse, and is never substantially worse than the best coordination mechanism.

We compare timestamp tokens against the Naiad-style notification API already available in Timely Dataflow. In order to compare with Flink-style watermarks without the confounding factor of running on a different platform (like Flink's), we re-implemented Flink's watermarks technique on the same communication and scheduling framework provided by Timely Dataflow. In some of the experiments (Figure 4.5, Figure 4.6, Figure 4.7), where the technique selected has limited impact on performance, timestamp tokens and Flink-style watermarks achieve nearly identical la-

tency, showing that our implementation does not unfairly disadvantage watermarks.

We observe that timestamp tokens avoid the collapse that notifications experience for high numbers of distinct timestamps (Figure 4.6, Figure 4.7), and the collapse that watermarks experience for complex dataflows (Figure 4.9, Figure 4.8). In all cases, timestamp tokens remain among the best approaches.

4.5.1 Experimental setup

We run all experiments on a CloudLab[Dup+19] server with one AMD EPYC 7452 with 32 physical cores and 128GB of RAM. We disable simultaneous multi-threading (SMT) and we pin each timely dataflow worker to a distinct physical core.

Our open-loop testing harness supplies the input at a specified rate, even if the system itself becomes less responsive. We record the observed latency in units of nanoseconds in a histogram of logarithmically-sized bins. If the system becomes overloaded and end-to-end latency becomes greater than 1 second, the testing harness regards the experiment as failed.

4.5.2 Microbenchmarks

Our microbenchmarks use a simple dataflow program that consists of a single stateful operator that computes the overall rolling count of unique words observed on the inputs. Every time the operator receives a word, it updates the internal count, and sends an output message with the updated value.

To determine the effectiveness of handling fine-grained timestamps with various techniques, we generate input at a given constant rate and assign different timestamps to each input tuple based on when it was generated. The assigned timestamps are quantized to powers-of-two ranging from 2^8 to 2^{16} nanoseconds ("ns" in the following). A timestamp quantum of 2^xns means that regardless of the input rate, there can be at most $\frac{1 \times 10^9}{2^x}$ distinct timestamps in the ingested data per second. For example, with a timestamp quantum of 2^8ns (256ns), at most 4 million timestamps per second can be generated.

Varying the size of the quantum allows us to evaluate how well a mechanism can handle coarser or finer timestamp granularities. With a smaller timestamp quantum, the system can provide higher time resolution in the output it produces. As previously discussed, with Naiad-style notifications, the operator needs to interact with the system for each logical time it processes, and for which it requires a notification.

4.5.2.1 Varying timestamp granularity

Figure 4.5 shows the achieved median, p999 (99.9%), and maximum latency when we vary the granularity of the timestamp quantization under different offered loads: 32 million tuples/sec is below the maximum throughput achievable with fine timestamp granularity by at least some of the coordination mechanisms, and 64 million tuples/sec represent a very high load that all mechanisms cannot sustain with a timestamp quantum of 2^{13}ns or finer. The performance pattern at lower loads is similar to what we report for 32 million tuples/sec, but with lower latency.

All mechanisms display similar performance characteristics when not overloaded, with two notable exceptions. First, notifications are unable to handle a timestamp granularity below 2^{13}ns; this is because they require an interaction between the operator logic and the system for each timestamp. That is not the case for both watermarks and tokens, that can handle any timestamp quantization. Second, the maximum latency for watermarks is 2x smaller than timestamp tokens for timestamp quantization above 2^{14}: for this extremely simple single-operator dataflow, watermarks can have slightly lower overhead at the tail.

At very high load (64 million tuples/sec) (i) all mechanisms have significantly higher tail latency and cannot handle the finest timestamp granularities, (ii) both watermarks and timestamp tokens can handle timestamp granularities finer than notifications, (iii) notifications achieve better p999 (when they are able to sustain the load) possibly due to additional synchronization imposed by the mechanism, and (iv) watermarks display slightly higher median latency at this load.

In this microbenchmark, timestamp tokens perform essentially on par with watermarks when not overloaded, and behave better when the system is overloaded. Notifications are unable to handle highly granular timestamps in the input data even at lower loads, because every timestamp requires an interaction between the operator logic and the system.

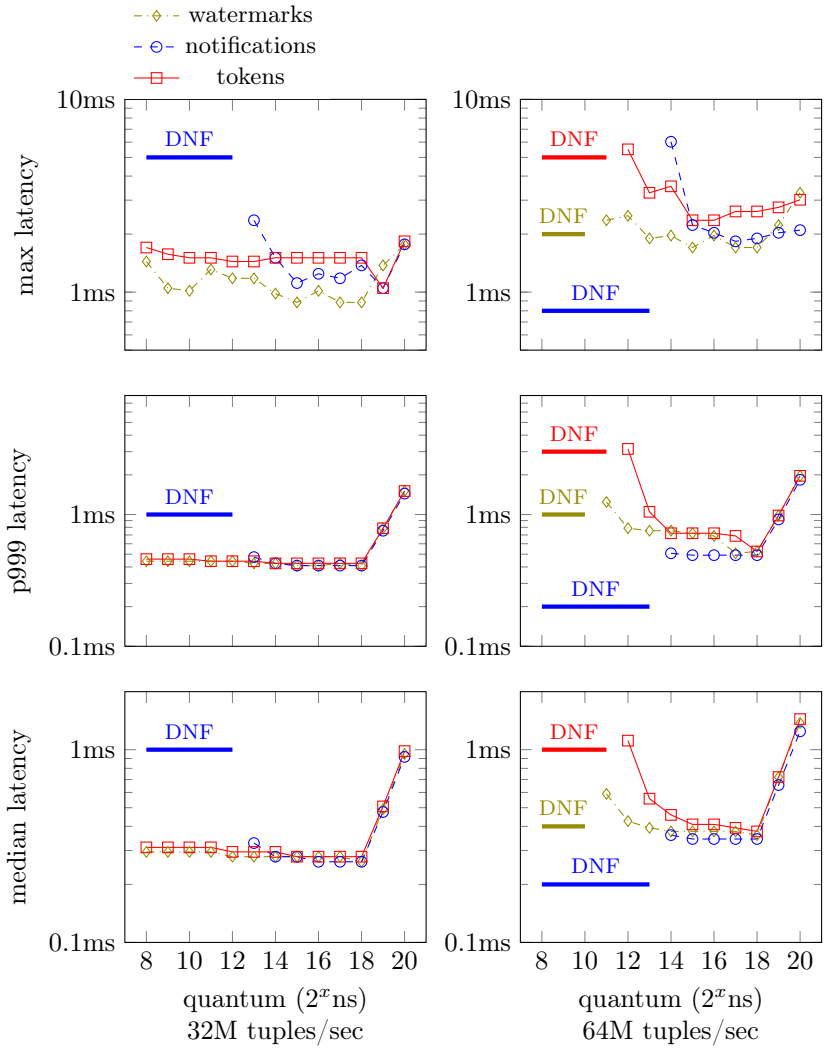


Figure 4.5: Latency for a single-operator ("word-count") dataflow with Flink-style watermarks, Naiad-style notifications, and with timestamp tokens. We run the workloads on 8 physical cores at three different offered loads. We report median, p999, and maximum latency as we vary the timestamp quantization. Note the different scales on the y axes of the plots.

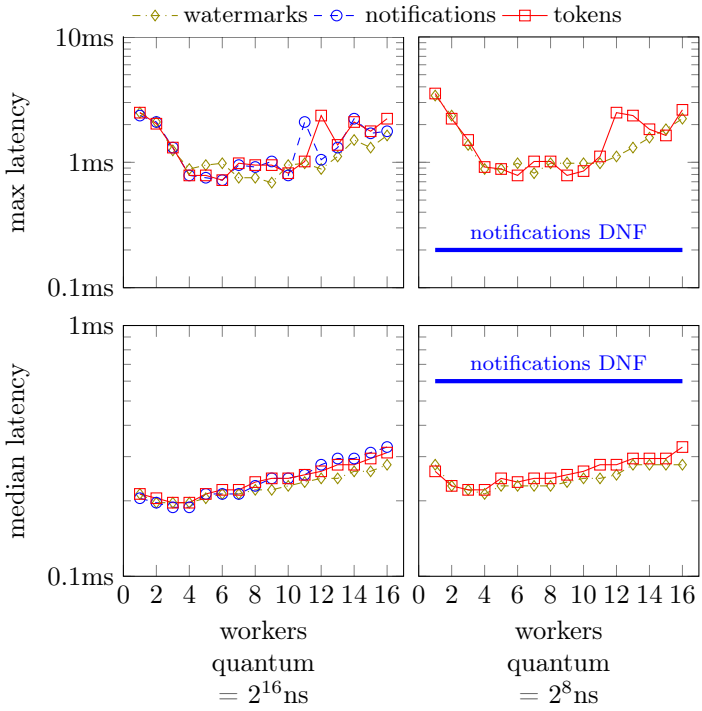


Figure 4.6: Weak scaling for the word-count workload. We report results with the timestamp quantization set to either 2^{16}ns or 2^{8}ns. We vary the number of workers and the offered load, which is fixed at 2 million tuples per second per worker. Note that Naiad-style notification fail to keep up with load for timestamp quantum = 2^{8}ns. Note the different scales on the y axes of the plots.

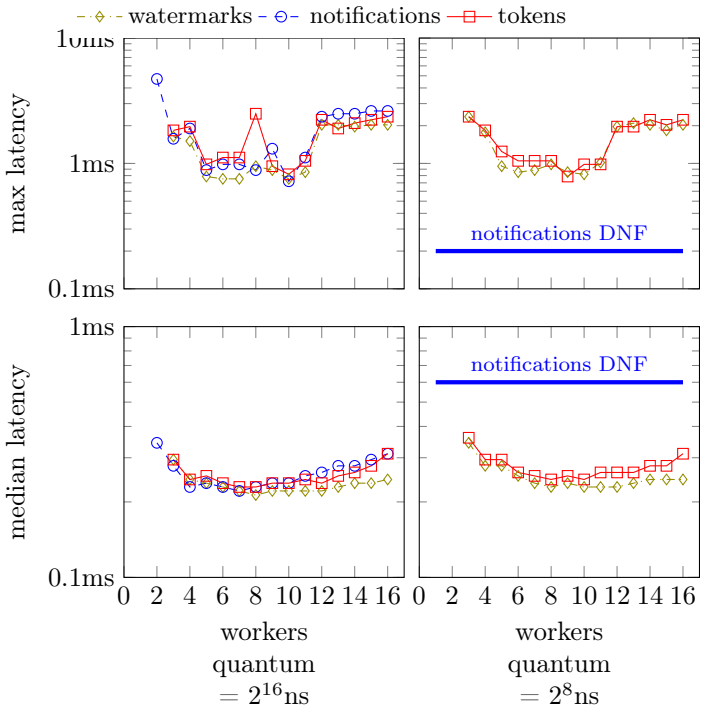


Figure 4.7: Strong scaling for the word-count workload. We report results with the timestamp quantization set to either 2^{16}ns or 2^{8}ns. We vary the number of workers while keeping the offered load fixed at 20 million tuples per second. For quantum = 2^{16}ns, notifications fail to keep up with load with less than 2 workers, and other mechanisms fail with less than 3 workers. For quantum = 2^{8}ns, all configurations fail with less than 3 workers, and notifications fail with any number of workers. Note the different scales on the y axes of the plots.

4.5.2.2 Scaling

Figure 4.6 and Figure 4.7 show the scaling behaviour of the microbenchmark word-count dataflow. At the coarser timestamp quantization granularity, all techniques display nearly identical scaling characteristics. In both strong and weak scaling we can see the system's and techniques' minor inefficiencies starting to affect the reported latency above around 6 workers. At the finer timestamp granularity, Naiad-style notifications fail to keep up with load at any scale, while watermarks and timestamp tokens display similar behaviour. This demonstrates that timestamp tokens do not negatively affect scaling.

4.5.3 Complex dataflow fragments

As discussed in subsection 4.4.2, timestamp tokens do not require continual interaction between the operator and the system to retire timestamps, in particular when an operator is idle for a period of time. To measure the performance benefit of not having to invoke each operator for each successive timestamp, even if no work needs to be performed, we construct a dataflow with a variable sequence of no-op operators (from 8 to 256 no-op operators connected as a sequential pipeline).

Timestamp tokens and Naiad-style notifications always calculate operator input frontiers (low watermarks) as if each channel between two consecutive operators may exchange data between workers. For Flink-style watermarks we need to distinguish between a scenario where a cross-worker exchange happens at each step (and watermarks are broadcast) and an additional (unrealistic) scenario at the other end of the spectrum where no cross-worker data exchange takes place. A real-world dataflow is likely to have a mix of worker-local and cross-worker channels, and would likely sit somewhere between these two extremes.

Figure 4.8 and Figure 4.9 show the performance impact of handling timestamps for a sequence of idle operators of varying length. Timestamp tokens, and Naiad-style notifications, and the Flink-style watermark configuration without cross-worker exchange (*watermarks-P*) have almost identical performance that is only marginally affected by the length of the operator chain (Figure 4.8) and by the workload scale (Figure 4.9). In this scenario *watermarks-P* has an unrealistic advantage because no coordination information is ever exchanged between workers: each processor

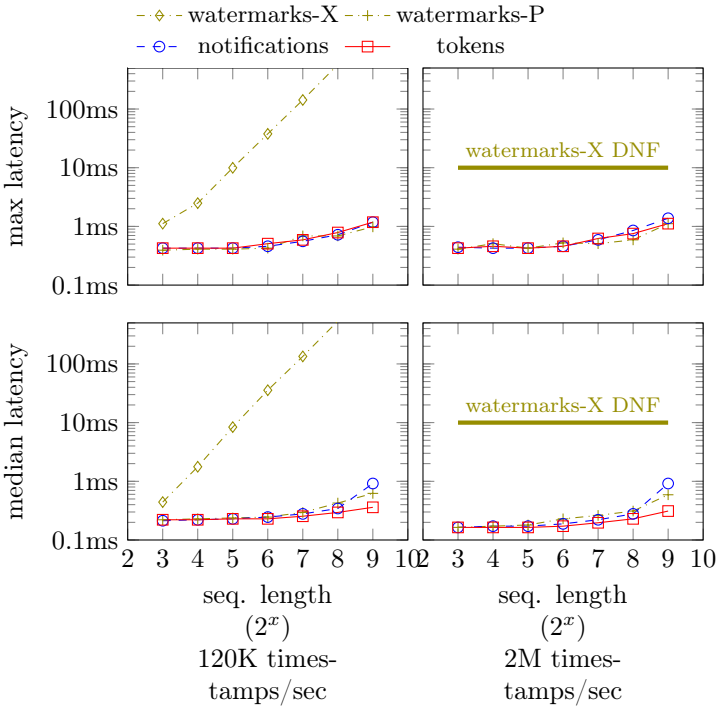


Figure 4.8: Impact of a long sequence of operators in the dataflow graph when varying the number of operators. For Flink-style watermarks we consider two dataflows: one with all-worker exchanges at every stage (watermarks-X) and one where operators form pipelines that are connected locally on each worker (watermarks-P). Note the different scales on the y axes of the plots. Sequence of no-op operators. We vary the number of operators in the sequence and the offered load in terms of timestamps/sec. We run the workloads on 8 physical cores.

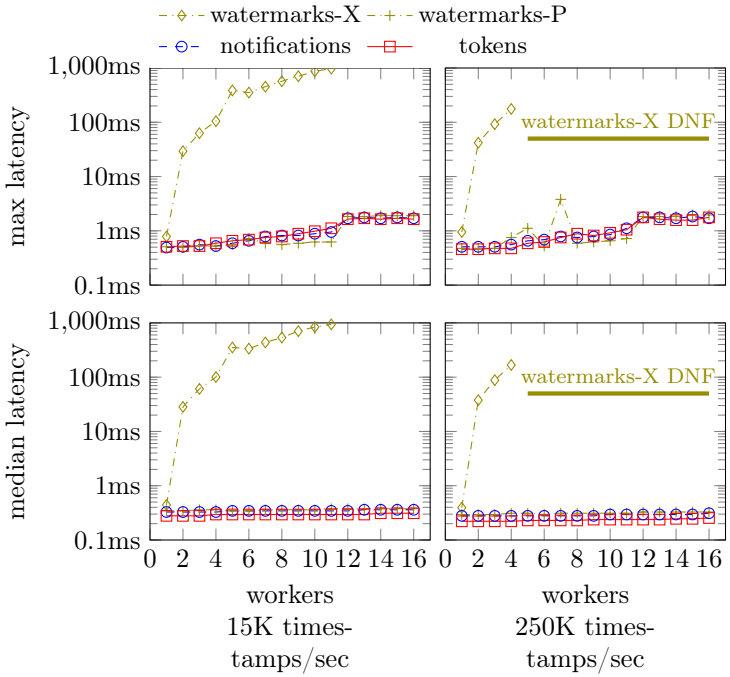


Figure 4.9: Impact of a long sequence of operators in the dataflow graph when varying the number of workers. For Flink-style watermarks we consider two dataflows: one with all-worker exchanges at every stage (watermarks-X) and one where operators form pipelines that are connected locally on each worker (watermarks-P). Note the different scales on the y axes of the plots. Weak scaling for an operator sequence of 256 no-op operators. We vary the number of workers while keeping the offered load fixed at 15K and 250K timestamps per second, per worker.

operates as a separate unit, and thus does not incur any coordination cost.

When configured to perform exchanges for every inter-operator channel (*watermarks-X*) the latency for Flink-style watermarks degrades linearly with the number of operators in the sequence (Figure 4.8) because each operator has to be invoked to forward the watermark which then needs to be broadcast to all other operators. This also fundamentally limits scalability: *watermarks-X* has to process watermarks proportional to the length of the sequence times the number of workers, resulting in high latency even at moderate scale.

By not requiring interaction with each operator for each timestamp, timestamp tokens matches or outperforms other techniques when handling complex inactive dataflow fragments.

4.5.4 NEXMark

To evaluate timestamp tokens' performance impact on a realistic, albeit simple, data processing use case, we extended the timely dataflow implementation of the NEXMark queries we open sourced as part of a related project [Hof+20]. The original implementation leverages timestamp tokens as described in Chapter 5. We augmented it by writing the same queries with Naiad-style notifications and Flink-style watermarks.

The NEXMark suite models an auction site in which a high-volume stream of users, auctions, and bids arrive, and standing queries are maintained reflecting a variety of relational queries. For the purpose of this experiment, we focus on queries that result in multi-operator dataflows (Q4 and Q7). Megaphone [Hof+19] describes the query semantics; for our purposes we only need to highlight that Q4 has a two-stage dataflow where one of the operators handles tokens to calculate a data-dependent windowed maximum, and Q7 has two stateful operators with two consecutive data exchanges.

Timestamp tokens avoid the collapse that notifications exhibit for Q4 due to overwhelming numbers of distinct timestamps, and are competitive with watermarks (improving on them slightly for Q7). These queries are relatively simple, only a few dataflow stages, and timestamp tokens do not have much room to distinguish themselves from watermarks.

NEXmark Q4		latency (milliseconds)								
		tokens			notifications			watermarks		
tuples/sec	workers	p50	p999	max	p50	p999	max	p50	p999	max
4M	4	0.62	1.25	1.9		DNF		0.25	0.59	1.25
4M	8	0.52	0.98	1.51		DNF		0.29	0.56	1.44
4M	12	0.59	1.02	5.77		DNF		0.38	0.56	2.49
6M	4		DNF			DNF			DNF	
6M	8	1.31	2.62	4.19		DNF		0.72	2.36	4.19
6M	12	1.25	2.36	2.88		DNF		0.51	1.02	3.54
8M	4		DNF			DNF			DNF	
8M	8		DNF			DNF			DNF	
8M	12	2.03	3.93	11.53		DNF		0.95	2.62	3.67

NEXmark Q7		latency (milliseconds)								
		tokens			notifications			watermarks		
tuples/sec	workers	p50	p999	max	p50	p999	max	p50	p999	max
4M	4	0.06	0.09	0.31	0.06	0.09	0.22	0.07	0.11	0.36
4M	8	0.06	0.1	0.46	0.06	0.09	0.41	0.08	0.13	0.66
4M	12	0.06	0.11	0.82	0.06	0.1	0.72	0.1	0.17	0.79
6M	4	0.06	0.1	0.23	0.06	0.1	0.38	0.07	0.11	0.26
6M	8	0.06	0.1	0.46	0.06	0.1	0.44	0.09	0.13	0.66
6M	12	0.07	0.11	0.92	0.06	0.11	0.95	0.11	0.18	0.82
8M	4	0.07	0.1	0.39	0.07	0.11	0.24	0.07	0.11	0.62
8M	8	0.07	0.11	0.56	0.06	0.1	0.44	0.09	0.15	0.69
8M	12	0.07	0.11	1.02	0.07	0.11	0.92	0.11	0.19	1.31

Table 4.1: End-to-end processing latency for NEXmark query 4 and query 6. We scale the number of workers while keeping the total load fixed at 4, 6, and 8 million tuples/sec. We report median, p999, and maximum latency in milliseconds. For Q4 note that Naiad-style notifications cannot sustain the load for any of the configurations and timestamp tokens and Flink-style watermarks cannot sustain higher loads with 4-8 workers.

4.6 Conclusions

We introduced timestamp tokens, a coordination primitive for dataflow systems. Timestamp tokens decouple the sophistication of operator scheduling logic from the task of system-wide coordination. Operators can add sophistication to their own implementations, including flow control, fine-grained timestamps, and optimized data structures. At the same time, timestamp tokens simplify the surrounding system, whose role in scheduling no longer needs to be the bottleneck it once was.

Looking forward, we think timestamp tokens have potential to drive other new dataflow programming idioms, without increasing system complexity. We are especially interested in timestamp tokens as dataflow breakpoints, and how holding timestamp tokens provides external agents the opportunity to suspend execution without fundamentally restructuring dataflow programs.

Finally, we've been delighted by the force multiplier of investing in general dataflow primitives. Many projects quickly and safely implemented new system behavior writing only application-level code. We should have more well-considered primitives and fewer systems.

5 Building with timestamp tokens

This chapter is based on *Timestamp tokens: a better coordination primitive for data-processing systems* [LM22]; some of the work presented in this chapter was initially reported as part of my Master Thesis "Programmable scheduling in a stream processing system" [Lat16].

Timestamp tokens have been in use for several years. In this chapter, we relate examples where we found timestamp tokens to be especially helpful in building frameworks that implement new dataflow programming patterns. In each case, timestamp tokens and specialized operator logic allowed projects to avoid re-implementing parts of the timely dataflow system itself.

5.1 Co-operative control flow

Dataflow operators may run for a long time or produce large amounts of output data, and should yield control so that other operators can execute and potentially retire some of the output data. However, Naiad's execution model asks an operator to run to completion for each notification, and the return of control is an indication that the operator has completed its task. Timestamp tokens allow operators to yield control without yielding the right to resume execution and produce output in the future.

My early prototype for a dataflow flow control mechanism, Faucet [LMC16; Lat16], uses timestamp tokens to implement *user-level flow control*. This mechanism supports dataflow operators that may produce unboundedly

large numbers of output messages for each input. Faucet operators produce outputs up to a certain limit and then yield control until these messages are retired. Whenever an operator yields due to a reached limit, it retains the timestamp token to indicate it has further output to produce. This design allowed me to implement flow control in user code, without requiring modifications to the underlying system.

5.2 Fine-grained timestamps

Systems that track real time may process events with timestamps denominated in nanoseconds. Naiad assumes responsibility for ordering all events with distinct timestamps, and for high-resolution timestamps this can overwhelm the system. Timestamp tokens provide a mechanism for the operator to determine the granularity at which it reports outstanding timestamps to the system, without involving the system in each timestamp that is processed.

In DD [McS+20], which we presented in detail in Chapter 6, each event has a potentially unique timestamp, and operators receive and must react to a stream of such events. Rather than present each timestamp to the timely dataflow system, DD's operator implementations batch messages into "intervals". An operator retains the least timestamp tokens for the times of un-batched messages it holds, and as the operator's frontier advances the operator creates new batches containing all events whose timestamps are not in advance of the new frontier. The operator uses its current timestamp tokens to produce any output corresponding to the batch, and then downgrades its timestamp tokens once, to the new lower envelope of its un-batched messages. This design allows the operators to interact with the host timely dataflow system at a coarse granularity, independent of the timestamp granularity.

5.3 Optimized scheduling

This section refers to joint work with the authors of "Megaphone: Latency-conscious state migration for distributed streaming dataflows" [Hof+19] which was included in Moritz Hoffmann's PhD thesis [Hof19]. In this section we

focus on the impact of timestamp tokens on the design and implementation of Megaphone.

Timely dataflow computations may act on general partially ordered timestamps, and with large numbers of outstanding events it may be unclear which events should be processed next. A system like Naiad stores all events in an unsorted list and performs a sequential pass through this list in each scheduling round, limiting the minimum latency. Alternately, stream processors that only act on totally ordered timestamps can use priority queues to quickly extract only the relevant events. Timestamp tokens provide operators the ability to organize their schedulable work themselves, without pushing their implementation into the system itself.

In Megaphone [Hof+19], a migration mechanism for timely dataflow, we implemented the NEXMark benchmark which contains a variety of streaming computations, and in particular a variety of *windowed* computations. These computations have timestamps that are denominated in nanoseconds, and in one case a windowed computation with a 12 hour continuous slide (and so, an effectively unbounded number of distinct timestamps in play at any time). Our implementation uses priority queues of timestamp tokens to schedule the work in these specific operators, providing millisecond latencies without compromising the ability of the rest of the system to handle partially-ordered timestamps.

6 Shared arrangements

This chapter is based on "Shared Arrangements: practical inter-query sharing for streaming dataflows" [McS+20] and the associated Technical Report [McS+18].

Current systems for data-parallel, incremental processing and view maintenance over high-rate streams isolate the execution of independent queries. This creates unwanted redundancy and overhead in the presence of concurrent incrementally maintained queries: each query must independently maintain the same indexed state over the same input streams, and new queries must build this state from scratch before they can begin to emit their first results.

This chapter introduces *shared arrangements*: indexed views of maintained state that allow concurrent queries to reuse the same in-memory state without compromising data-parallel performance and scaling. We implement shared arrangements in a modern time-aware dataflow processor and show order-of-magnitude improvements in query response time and resource consumption for interactive queries against high-throughput streams, while also significantly improving performance in other domains including business analytics, graph processing, and program analysis.

6.1 Introduction

In this chapter, we present *shared arrangements*, a new technique for efficiently sharing indexed, consistent state and computation between the operators of multiple concurrent, data-parallel streaming dataflows. We have implemented shared arrangements in DD [Mur+13; McS+13; MD], but they are broadly applicable to other time-aware dataflow systems.

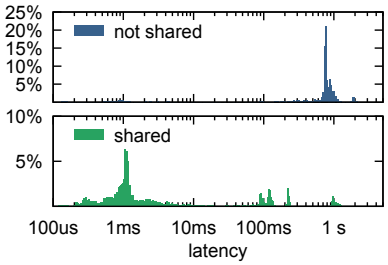
Shared arrangements are particularly effective in interactive data analytics against continually-updating data. Consider a setting in which multiple analysts, as well as software like business intelligence dashboards and monitoring systems, interactively submit standing queries to a stream processing system. The queries remain active until they are removed. Ideally, queries would install quickly, provide initial results promptly, and continue to deliver updates with low latency as the underlying data change.

Data-parallel stream processors like Flink [Car+15], Spark Streaming [Zah+13], and Naiad [Mur+13] excel at incrementally maintaining the results of such queries, but each maintain queries in independent dataflows with independent computation and operator state. Although these systems support the sharing of common sub-queries, as streams of data, none share the *indexed* representations of relations among unrelated subqueries.

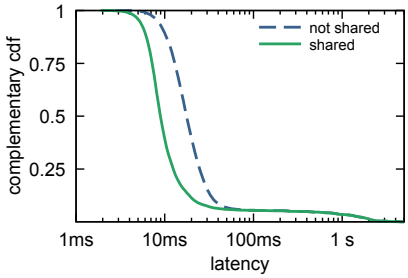
However, there are tremendous opportunties for sharing of state, even when the dataflow operators are not the same. For example, we might expect joins of a relation R to use its primary key; even if several distict queries join R against as many other distinct relations, a shared index on R would benefit each query. Existing systems create independent dataflows for distinct queries, or are restricted to redundant, per-query indexed representations of R, wasting memory and computation.

By contrast, classic relational databases have long shared indexes over their tables across unrelated queries. The use of shared indexes reduces query times tremendously, especially for point look-ups, and generally improves the efficiency of queries that access relations by the index keys. While they have many capabilities, relational databases lack streaming dataflow system's support for low-latency, high-throuhput incremental maintenance of materialized query results [Gje+18; Ahm+12]. Existing shared index implementations share all reads and writes among multiple workers, and are not immediately appropriate for dataflow workloads where the operator state is sharded across independent workers. In this work, we seek to transport the shared index idiom from relational databases to streaming dataflows, applying it across changing maintained queries.

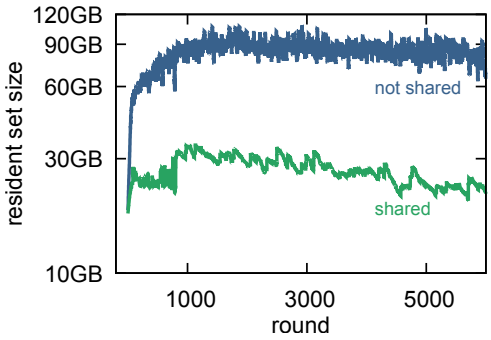
Our main observations are that *(i)* many dataflow operators write the same internal state, representing the accumulated changes of each of their input streams, *(ii)* these dataflow operators often access this state with



(a) Query installation latency.



(b) Update processing latency.



(c) Memory footprint (RSS).

Figure 6.1: Shared arrangements reduce ((a)) query installation latency distribution, ((b)) update processing latency distribution, and ((c)) the memory footprint of concurrent TPC-H queries that randomly arrive and retire. The setup uses 32 workers, runs at TPC-H scale factor 10, and loads rows from relations round-robin. Note the $\log_{10}$-scale x-axes in ((a)) and ((b)), and the $\log_{10}$-scale y-axis in ((c)).

independent and fundamentally different patterns, and *(iii)* this state can be efficiently shared with single-writer, multiple-reader data structure. Shared arrangements are our design for single-writer, multiple-reader, shared state in dataflow systems.

To illustrate a natural setting for shared arrangements, we run a mix of interactively issued and incrementally maintained TPC-H [TPC] queries executed as dataflows against a stream of order fulfillment events (i.e. changes to the `lineitem` relation). This is similar to a modern business analytics setting with advertisers, impressions, and advertising channels, and our dynamic query setup mimicks the behavior of human analysts and business analytics dashboards.[1] We measure the query installation latency—i.e. the time until a new query returns results—as well as update processing latency and standing memory footprint. Figure 6.1 reports the performance of DD with shared arrangements ("shared") and without ("not shared"; representative of other data-parallel stream processors). The measurements show orders of magnitude improvements in query installation latency (a weakness of existing dataflow systems), and improved update processing latency and memory use.

Shared arrangements achieve these improvements because they remove the need to maintain dataflow-local indexes for each query. As a concrete example throughout this chapter, we consider TPC-H queries 3 and 5. Both queries join `lineitem` with the `order` and `customer` relations by their primary keys. While the queries lack overlapping subqueries that classic multi-query optimization (MQO) would detect, they both perform lookups into `order` and `customer` by their respective primary keys when processing an updated `lineitem` record. Existing stream processors will create and maintain a per-query index for each relation, as these systems are designed to decouple the execution of dataflow operators. Shared arrangements, by contrast, allow Q3 and Q5 to share indexes for these two relations. This can dramatically reduce the time to install the second query and provide initial results, and also increases overall system capacity, as multiple queries share in-memory indexes over the same relations. Finally, these benefits come without restricting update throughput or latency, as they do not change the data-parallel execution model of the stream processor.

[1] TPC-H is originally a static "data-warehousing" benchmark; our streaming setup follows that used by Nikolic et al. [NDK16].

The key challenge for shared arrangements is to balance the opportunities of sharing against the need for coordination in the execution of the dataflow. In the scenarios we target, logical operator state is sharded across multiple physical operators; sharing this state between the operators of multiple queries could require global synchronization. Arrangements solve this challenge by carefully structuring how they share data: they *(i)* hard-partition shared state between worker threads and move computation (operators) to it, and *(ii)* multiversion shared state within workers to allow operators to interact with it at different times and rates.

Our full results in section 6.7 confirm that shared arrangements translate into two benefits: *(i)* queries deploy and produce correct results immediately without rescanning historical data, and *(ii)* the same capacity (stream volume and concurrent queries) can be achieved with fewer cores and less RAM. For a streaming variant of TPC-H and a changing graph, shared arrangements also reduce update latency by 1.3–3× and reduce the memory footprint of the computation by 2–4×, compared to systems that do not share indexed state. These benefits hold without degrading performance on other tasks—batch and interactive graph processing, and Datalog-based program analysis—on which DD outperforms other systems.

Shared arrangements can be applied to many modern time-aware dataflow systems, but we implemented them as part of DD. DD has been the publicly available reference implementation of Differential Dataflow for several years [MD], and is deployed in variety of industrial settings. For example, VMware Research uses DD to back their reactive DDlog Datalog engine [RB19], applied to problems in network reconfiguration and program analysis. Shared arrangements have proved key to the system's success.

Some benefits of shared arrangements are attainable in purely windowed streaming settings, which ensure that only bounded historical state must be reviewed for new queries. However, shared arrangements provide similar benefits without these restrictions, and support windowing of data as one of several join idioms. The main limitation of shared arrangements is that their benefits apply only in the cases where actual sharing occurs; while sharing appears common in settings with relational data and queries, bespoke stream processing computations (e.g. with complex and disjoint windowing on relations) may benefit to varying and lesser degrees.

In many ways, shared arrangements are the natural interpretation of an RDBMS index for data-parallel dataflow, and bring its benefits to a domain that has until now lacked them.

6.2 Background and Related Work

Shared arrangements allow queries to share indexed state. Inter-query state sharing can be framed in terms of *(i) what* can be shared between queries, *(ii)* if this shared state can be *updated*, and *(iii)* the *coordination* required to maintain it. Figure 6.1 compares sharing in different classes of systems.

Relational databases like PostgreSQL [Pos] excel at answering queries over schema-defined tables. Indexes help them speed up access to records in these tables, turning sequential scans into point lookups. When the underlying records change, the database updates the index. This model is flexible and shares indexes between different queries, but it requires coordination (e.g. locking [Dar19]). Scaling this coordination out to many parallel processors or servers holding shards of a large database has proven difficult, and scalable systems consequently restrict coordination.

Parallel-processing **"big data" systems** like MapReduce [DG04], Dryad [Isa+07], and Spark [Zah+12] rely only on coarse-grained coordination. They avoid indexes and turn query processing into parallel scans of distributed collections. But these collections are immutable: any change to a distributed collection (e.g. a Spark RDD) requires reconstituting that collection as a new one. This captures a collection's lineage and makes all parallelism deterministic, which eases recovery from failures. Immutability allows different queries to share the (static) collection for reading [Gun+10]. This design aids scale-out, but makes these systems a poor fit for streaming computations, with frequent fine-grained changes to the collections.

Stream-processing systems reintroduce fine-grained mutability, but they lack sharing. Systems like Flink [Car+15], Naiad [Mur+13], and Noria [Gje+18] keep long-lived, indexed intermediate results in memory for efficient incremental processing, partitioning the computation across workers for scale-out, data-parallel processing. However, stream processors associate each piece of state *exclusively* with a single operator, since concurrent accesses to this state from multiple operators would race with

System class	Example	Sharing	Updates	Coordination
RDBMS	Postgres	**Indexed state**	**Record-level**	Fine-grained
Batch processor	Spark	Non-indexed collections	Whole collection	**Coarse-grained**
Stream processor	Flink	None	**Record-level**	**Coarse-grained**
Shared arrangements	DD	**Indexed state**	**Record-level**	**Coarse-grained**

Table 6.1: Sharing of indexed in-memory state, record-level update granularity, and scalability through coarse-grained coordination are not all found in current systems. Shared arrangements combine these features in a single system.

state mutations. Consequently, these systems *duplicate* the state that operators could, in principle, share.

By contrast, **shared arrangements** allow for fine-grained updates to shared indexes and preserve the scalability of data-parallel computation. In particular, shared arrangements rely on multiversioned indices and data-parallel sharding to allow updates to shared state without the costly coordination mechanisms of classic databases. In exchange for scalability and parallelism, shared arrangements give up some abilities. Unlike indexes in relational databases, shared arrangements do not support multiple writers, and are not suitable tools to implement a general transaction processor. Because sharing entangles queries that would otherwise execute in isolation, it can reduce performance and fault isolation between queries compared to redundant, duplicated state.

It is important to contrast shared arrangements to Multi-Query Optimization (MQO) mechanisms that identify overlapping subqueries. MQO shares state and processing between queries with common subexpressions, but shared arrangements also benefit distinct queries that access the same indexes. Both relational and big data systems can identify common sub-expressions via MQO and either cache their results or fuse their computation. For example, CJoin [CPV09] and SharedDB [GAK12] share table scans between concurrent ad-hoc queries over large, unindexed tables in data warehousing contexts, and Nectar [Gun+10] does so for DryadLINQ [Yu+08] computations. More recently AStream [KRM19] applied the architecture of SharedDB to windowed streaming computation, and can share among queries the resources applied to future windows. TelegraphCQ [Cha03] and DBToaster [Ahm+12] share state among continuous queries, but sequentially process each query without parallelism or shared indexes. Noria [Gje+18] shares computation between queries over streams, but again lacks shared indexes. In all these systems, potential sharing must be identified at query deployment time; none provide new queries with access to indexed historical state. In constrast, shared arrangements (like database indices) allow for post-hoc sharing: new queries can immediately attach to the in-memory arrangements of existing queries, and quickly start producing correct outputs that reflect all prior events.

Philosophically closest to shared arrangements is STREAM [BW01], a relational stream processor which maintains "synopses" (often indexes) for operators and shares them between operators. In contrast to shared ar-

rangements, STREAM synopses lack features necessary for coarse-grained data-parallel incremental view maintenance: STREAM synopses are not multiversioned and do not support sharding for data-parallelism. STREAM processes records one-at-a-time; shared arrangements expose a stream of shared, indexed batches to optimized implementations of the operators.

Shared arrangements allow for operators fundamentally designed around shared indexes. Their ideas are, in principle, compatible with many existing stream processors that provide versioned updates (as e.g. Naiad and Flink do) and support physical co-location of operator shards (as e.g. Naiad and Noria do).

6.3 Shared Arrangements Overview

The high-level objective of shared arrangements is to share indexed operator state, both within a single dataflow and across multiple concurrent dataflows, serving concurrent continuous queries. Shared arrangements substitute for per-instance operator state in the dataflow, and should appear to an individual operator as if it was a private copy of its state. Across operators, the shared arrangement's semantics are identical to maintaining individual copies of the indexed state in each operator. At the same time, the shared arrangement permits index reuse between operators that proceed at a different pace due to asynchrony in the system.

Operators that provide incremental view maintenance, so that their output continually reflects their accumulated input updates, offer particularly good opportunities for sharing state. This is because each stream of updates has one logical interpretation: as an accumulation of all updates. When multiple such operators want to build the same state, but vary what subset to read based on the time t they are currently processing, they can share arrangements instead. We assume that developers specify their dataflows using existing interfaces, but that they (or an optimizing compiler) explicitly indicate which dataflow state to share among which operators.

A shared arrangement exposes different *versions* of the underlying state to different operators, depending on their current time t. The arrangement therefore emulates, atop physically shared state, the separate indexes that operators would otherwise keep. Specifically, shared arrangements maintain state for operators whose state consists of the input col-

Collection trace

```
(data=(id=342, "Company LLC", "USA"),         time=4350, diff=+1)
(data=(id=563, "Firma GmbH", "Deutschland"), time=4355, diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      time=4360, diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      time=6200, diff=-1)
(data=(id=225, "Company Ltd", "UK"),          time=6220, diff=+1)
```

Collection at time $t = 4360$

```
(data=(id=342, "Company LLC", "USA"),         diff=+1)
(data=(id=563, "Firma GmbH", "Deutschland"), diff=+1)
(data=(id=225, "Azienda SRL", "Italia"),      diff=+1)
```

Collection at time $t = 6230$

```
(data=(id=342, "Company LLC", "USA"),    diff=+1)
(data=(id=563, "Firma", "Deutschland"), diff=+1)
(data=(id=225, "Company Ltd", "UK"),     diff=+1)
```

Figure 6.2: Update triples incoming to an operator, a "collection trace", and the resulting collection view at different times.

lection (i.e. the cumulative streaming input). Following Differential Dataflow [MD] terminology, a *collection trace* is the set of update triples *(data, time, diff)* that define a collection at time t by the accumulation of those inputs *(data, diff)* for which *time* $\leqslant t$ (Figure 6.2). Each downstream operator selects a different view based on a different time t of accumulation. Formal semantics of differential dataflow operators are presented in [AMP15].

An explicit, new `arrange` operator maintains the multiversioned state and views, while downstream operators read from their respective views. The contents of these views vary according to current logical timestamp frontier at the different operators: for example, a downstream operator's view may not yet contain updates that the upstream `arrange` operator has already added into the index for a future logical time if the operator has yet to process them.

Downstream operators in the same dataflow, and operators in other dataflows operating in the same logical time domain, can share the arrangement as long as they use the same key as the arrangement index. In particular, sharing can extend as far as the next change of key (an `exchange` operator in Differential Dataflow, or a 'shuffle' in Flink), an arrangement-unaware operator (e.g. `map`, which may change the key), or an operator that explicitly materializes a new collection.

6.3.1 Shared Arrangements Example

We illustrate a concrete use of shared arrangements with the example of TPC-H Q3 and Q5. Recall that in our target setting, analysts author and execute SQL queries against heavily normalized datasets. Relations in analytics queries are commonly normalized into "fact" and "dimension" tables, the former containing foreign keys into the latter. While new facts (e.g. ad impressions, or line items in TPC-H) are continually added, the dimension tables are also updated (for example, when a customer or supplier updates their information). The dimension tables are excellent candidates for arrangement by primary keys: we expect many uses of these tables to be joins by primary keys, and each time this happens an arrangement can be shared rather than reconstructed.

TPC-H Q3 retrieves the ten unshipped orders with the highest value. This is a natural query to maintain, as analysts work to unblock a potential backlog of valuable orders. The query derives from three relations—`lineitem`, `orders`, and `customer`—joined using the primary keys on `orders` and `customer`. A dataflow would start from `lineitem` and join against `orders` and `customer` in sequence. TPC-H Q5 lists the revenue volume done through local suppliers, and derives from three more relations (`supplier`, `nation`, and `region`). Each relation other than `lineitem` is joined using its primary key. A dataflow might start from `lineitem` and join against dimension tables in a sequence that makes a foreign key available for each table before joining it. In both queries, each dimension table is sharded across workers by their primary key.

The two queries do not have overlapping subqueries—each has different filters on order dates, for example—but both join against `orders` and `customer` by their primary keys. Deployed on the same workers, we first apply `arrange` operators to the `orders` and `customer` relations by their primary keys, shuffling updates to these relations by their key and resulting in shareable arrangements. In separate dataflows, Q3 and Q5 both have `join` operators that take as input the corresponding arrangement, rather than the streams of updates that formed them. As each arrangement is pre-sharded by its key, each worker has only to connect its part of each arrangement to its dataflow operators. Each worker must still stream in the `lineitem` data but the time for the query to return results becomes independent of the sizes of `orders` and `customer`.

6.3.2 System Features Supporting Efficiency

Shared arrangements apply in the general dataflow setting described in Chapter 3, and can benefit any system with those properties. But additional system properties can make an implementation more performant. We base our implementation on frameworks (Timely and Differential Dataflow) with these properties.

Timestamp batches Timestamps in Timely Dataflow only need to be *partially ordered*. The partial order of these timestamps allows Timely Dataflow graphs to avoid unintentional concurrency blockers, like serializing the execution of rounds of input (Spark) or rounds of iteration (Flink). By removing these logical concurrency blockers, the system can retire larger groups of logical times at once, and produce larger batches of updates. This benefits DD because the atoms of shared state can increase in granularity, and the coordination between the sharing sites can decrease substantially. Systems that must retire smaller batches of timestamps must coordinate more frequently, which can limit their update rates.

Multiversioned state Differential Dataflow has native support for *multiversioned* state. This allows it to work concurrently on any updates that are not yet beyond the Timely Dataflow frontier, without imposing a serial execution order on updates. Multiversioned state benefits shared arrangements because it decouples the execution of the operators that share the state. Without multiversioned state, operators that share state must have their executions interleaved for each logical time, which increases coordination.

Co-scheduling Timely Dataflow allows each worker to host an unbounded number of dataflow operators, which the worker then schedules. This increases the complexity of each worker compared to a system with one thread per dataflow operator, but it increases the efficiency in complex dataflows with more operators than system threads. Co-scheduling benefits shared arrangements because the state shared between operators can be partitioned between worker threads, who do not need mutexes or locks to manage concurrency. Systems that cannot co-schedule operators

that share state must use inter-thread or inter-process mechanisms to access shared state, increasing complexity and the cost.

Incremental Updates Differential dataflow operators are designed to provide incremental view maintenance: their output updates continually reflects their accumulated input updates. This restriction from general-purpose stream processing makes it easier to compose dataflows based on operators with clear sharing semantics. Systems that provide more general interfaces, including Timely Dataflow, push a substantial burden on to the user to identify operators that can share semantically equivalent state.

6.4 Implementation

Our implementation of a shared arrangement consists of three inter-related components:

1. the *trace*, a list of immutable, indexed batches of updates that together make up the multiversioned index;
2. an `arrange` operator, which mints new batches of updates, and writes them to and maintains the trace; and
3. read *handles*, through which arrangement-aware operators access the trace.

Each shared arrangement has its updates partitioned by the key of its index, across the participating dataflow workers. This same partitioning applies to the trace, the `arrange` operator, and the read handles, each of whose interactions are purely *intra*-worker; each worker maintains and shares its *shard* of the whole arrangement. The only inter-worker interaction is the pre-shuffling of inbound updates which effects the partition.

Figure 6.3 depicts a dataflow which uses an arrangement for the `count` operator, which must take a stream of *(data, time, diff)* updates and report the changes to accumulated counts for each *data*. This operation can be implemented by first partitioning the stream among workers by *data*, after which each worker maintains an index from *data* to its history, a list of *(time, diff)*. This same indexed representation is what is needed

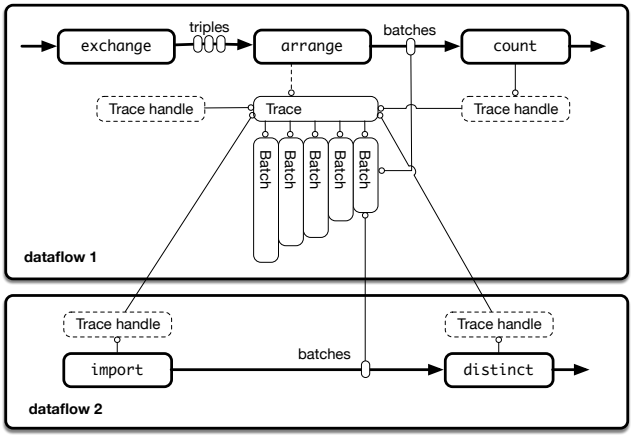


Figure 6.3: A worker-local overview of arrangement. Here the arrangement is constructed for the `count` operator, but is shared with a `distinct` operator in another dataflow. Each other worker performs the same collective data exchange, followed by local batch creation, trace maintenance, and sharing.

by the `distinct` operator, in a second dataflow, which can re-use the same partitioned and indexed arrangement rather than re-construct the arrangement itself.

6.4.1 Collection traces

As in Differential Dataflow, a *collection trace* is the set of update triples *(data, time, diff)* that define a collection at any time t by the accumulation of those *(data, diff)* for which $time \leqslant t$. A collection trace is initially empty and is only revealed as a computation proceeds, determined either as an input to the dataflow or from the output of another dataflow operator. Although update triples may arrive continually, it is only when the Timely Dataflow input frontier advances that the `arrange` operator learns that the updates for a subset of times are now complete.

In our design a collection trace is logically equivalent to an append-only list of immutable batches of update triples. Each batch is described by two frontiers of times, *lower* and *upper*, and the batch contains exactly

those updates whose times are beyond the lower frontier and not beyond of the upper frontier. The upper frontier of each batch matches the lower frontier of the next batch, and the growing list of batches reports the developing history of confirmed updates triples. A batch may be empty, which indicates that no updates exist in the indicated range of times.

To support efficient navigation of the collection trace, each batch is indexed by its *data* to provide random access to the history of each *data* (the set of its (*time*, *diff*) pairs). Background merge computation (performed by the `arrange` operator) ensures that at any time, a trace consists of logarithmically many batches, which ensures that operators can efficiently navigate the union of all batches.

Each reader of a trace holds a *trace handle*, which acts as a cursor that can navigate the multiversioned index. Each handle has an associated frontier, and ensures that it provides correct views of the index for any times beyond this frontier. Trace readers advance the frontier of their trace handle when they no longer require certain historical distinctions, which allows the `arrange` operator to compact batches by coalescing updates at older times, and to maintain a bounded memory footprint as a collection evolves.

6.4.2 The `arrange` operator

The `arrange` operator receives update triples, and must both create new immutable indexed batches of updates as its input frontier advances and compactly maintain the collection trace without violating its obligations to readers of the trace.

At a high level, the `arrange` operator buffers incoming update triples until the input frontier advances, at which point it extracts and indexes all buffered updates not beyond the newly advanced input frontier. A shared reference to this new immutable batch is both added to the trace and emitted as output from the `arrange` operator. When adding the batch to the trace, the operator may need to perform some maintenance to keep the trace representation compact and easy to navigate.

Batch implementation Each batch is immutable, but indexed to provide efficient random access. Our default implementation sorts update triples *(data, time, diff)* first by *data* and then by *time*, and stores the fields each in its own column. This balances the performance of read

latency, read throughput, and merge throughput. We have other batch implementations for specific domains (e.g. graphs), and new user implementations can be added without changing the surrounding superstructure. Most OLTP index structures are more general than needed for our immutable batches, but many of their data layout ideas could still be imported.

Amortized trace maintenance The maintenance work of merging batches in a trace is amortized over the introduced batches, so that no batch causes a spike in computation (and a resulting spike in latency). Informally, the operator performs the same *set* of merges as would a merge sort applied to the full sequence of batches, but only as the batches become available. Additionally, each merge is processed in steps: for each new batch, we perform work proportional to the batch size on each incomplete merge. A higher constant of proportionality leads to more eager merging, improving the throughput of the computation, whereas a lower constant improves the maximum latency of the computation.

Consolidation As readers of the trace advance through time, historical times become indistinguishable and updates at such times to the same *data* can be coalesced. The logic to determine which times are indistinguishable is present in Naiad's prototype implementation [Mur+13], but the mathematics of compaction have not been reported previously. In section 6.6, we present proofs of optimality and correctness.

Shared references Both immutable batches and traces themselves are reference counted. Importantly, the `arrange` operator holds only a "weak" reference to its trace, and if all readers of the trace drop their handles the operator will continue to produce batches but cease updating the trace. This optimization is crucial for competitive performance in computations that use both dynamic and static collections.

6.4.3 Trace handles

Read access to a collection trace is provided through a *trace handle*. A trace handle provides the ability to `import` a collection into a new dataflow, and to manually navigate a collection, but both only "as of" a re-

stricted set of times. Each trace handle maintains a frontier, and guarantees only that accumulated collections will be correct when accumulated to a time beyond this frontier. The trace itself tracks outstanding trace handle frontiers, which indirectly inform it about times that are indistinguishable to all readers (and which can therefore be coalesced).

Many operators (including `join` and `group`) only need access to their accumulated input collections for times beyond their input frontiers. As these frontiers advance, the operators are able to advance the frontier on their trace handles and still function correctly. The `join` operator is even able to drop the trace handle for an input when its *other* input ceases changing. These actions, advancing the frontier and dropping trace handles, provide the `arrange` operator with the opportunity to consolidate the representation of its trace, and in extreme cases discard it entirely.

A trace handle has an `import` method that, in a new dataflow, creates an arrangement exactly mirroring that of the trace. The imported collection immediately produces any existing consolidated historical batches, and begins to produce newly minted batches. The historical batches reflect all updates applied to the collection, either with full historical detail or coalesced to a more recent timestamp, depending on whether the handle's frontier has been advanced before importing the trace. Computations require no special logic or modes to accommodate attaching to in-progress streams; imported traces appear indistinguishable to their original streams, other than their unusually large batch sizes and recent timestamps.

6.5 Arrangement-aware operators

Operators act on collections, which can be represented either as a stream of update triples or as an arrangement. These two representations lead to different operator implementations, where the arrangement-based implementations can be substantially more efficient than traditional record-at-a-time operator implementations. In this section we explain arrangement-aware operator designs, starting with the simplest examples and proceeding to the more complex `join`, `group`, and `iterate` operators.

6.5.1 Key-preserving stateless operators

Several stateless operators are "key-preserving": they do not transform their input data to the point that it needs to be re-arranged. Example operators are `filter`, `concat`, `negate`, and the iteration helper methods `enter` and `leave`. These operators are implemented as streaming operators for streams of update triples, and as wrappers around arrangements that produce new arrangements. For example, the `filter` operator results in an arrangement that applies a supplied predicate as part of navigating through a wrapped inner arrangement. This design implies a trade-off, as an aggressive filter may reduce the data volume to the point that it is cheap to maintain a separate index, and relatively ineffective to search in a large index only to discard the majority of results. The user controls which implementation to use: they can filter an arrangement, or reduce the arrangement to a stream of updates and then filter it.

6.5.2 Key-altering stateless operators

Some stateless operators are "key-altering", in that the indexed representation of their output has little in common with that of their input. One obvious example is the `map` operator, which may perform arbitrary record-to-record transformations. These operators always produce outputs represented as streams of update triples.

6.5.3 Stateful operators

Differential Dataflow's stateful operators are data-parallel: their input *data* have a *(key, val)* structure, and the computation acts independently on each group of *key* data. This independence is what allows Naiad and similar systems to distribute operator work across otherwise independent workers, which can then process their work without further coordination. At a finer scale, this independence means that each worker can determine the effects of a sequence of updates on a key-by-key basis, resolving all updates to one key before moving to the next, even if this violates timestamp order.

6.5.3.1 The `join` operator

Our `join` operator takes as inputs batches of updates from each of its arranged inputs. It produces any changes in outputs that result from its advancing inputs, but our implementation has several variations from a traditional streaming hash-join.

Trace capabilities The join operator is bi-linear, and needs only each input trace in order to respond to updates from the *other* input. As such, the operator can advance the frontiers of each trace handle by the frontier of the other input, and it can drop each trace handle when the other input closes out. This is helpful if one input is static, as in iterative processing of static graphs.

Alternating seeks Join can receive input batches of substantial size, especially when importing an existing shared arrangement. Naively implemented, we might require time linear in the input batch sizes. Instead, we perform alternating seeks between the cursors for input batches and traces of the other input: when the cursor keys match we perform work, and if the keys do not match we seek forward for the larger key in the cursor with the smaller key. This pattern ensures that we perform work at most linear in the smaller of the two sizes, seeking rather than scanning through the cursor of the larger trace, even when it is supplied as an input batch.

Amortized work The `join` operator may produce a significant amount of output data that can be reduced only once it crosses an exchange edge for a downstream operator. If each input batch is immediately processed to completion, workers may be overwhelmed by the output, either buffered for transmission or (as in our prototype) sent to destination workers but buffered at each awaiting reduction. Instead, operators respond to input batches by producing "futures", limited batches of computation that can each be executed until sufficiently many outputs are produced, and then suspend. Futures make copies of the shared batch and trace references they use, which avoids blocking state maintenance for other operators.

6.5.3.2 The `group` operator

The `group` operator takes as input an arranged collection with data of the form *(key, val)* and a reduction function from a key and list of values to a list of output values. At each time the output might change, we reform the input and apply the reduction function, and compare the results to the reformed output to determine if output changes are required.

Perhaps surprisingly, the output may change at times that do not appear in the input (as the least upper bound of two times does not need to be one of the times). Hence, the `group` operator tracks a list of pairs *(key, time)* of future work that are required even if we see no input updates for the key at that time. For each such *(key, time)* pair, the `group` operator accumulates the input and output for *key* at *time*, applies the reduction function to the input, and subtracts the accumulated output to produce any corrective output updates.

Output arrangements The `group` operator uses a shared arrangement for its output, to efficiently reconstruct what it has previously produced as output without extensive re-invocation of the supplied user logic (and to avoid potential non-determinism therein). This provides the `group` operator the opportunity to share its output trace, just as the `arrange` operator does. It is common, especially in graph processing, for the results of a `group` to be immediately joined on the same key, and `join` can re-use the same indexed representation that `group` uses internally for its output.

6.5.4 Iteration

The iteration operator is essentially unchanged from Naiad's Differential Dataflow implementation. We have ensured that arrangements can be brought in to iterative scopes from outer scopes using only an arrangement wrapper, which allows access to shared arrangements in iterative computations.

6.6 Compaction Theorems

This section presents proof of optimality and correctness of trace compaction, introduced in subsection 6.4.2 in the paragraph "Consolidation".

Let F be an antichain of partially ordered times (a "frontier"). Writing $f \geqslant F$ to mean f is greater than some element of F, we will say that two times t_1 and t_2 are "indistinguishable as of F", written $t_1 \equiv_F t_2$, when

$$t_1 \equiv_F t_2 \text{ when } \forall_{f \geqslant F}(t_1 \leqslant f \text{ iff } t_2 \leqslant f)$$

As performed in the Naiad prototype, we can determine a representative for a time t relative to a set F using the least upper bound ($\wedge$) and greatest lower bound ($\vee$) operations of the lattice of times, taking the greatest lower bound of the set of least upper bounds of t_i and elements of F:

$$rep_F(t) := \vee_{f \in F}(t \wedge f)$$

The rep_F function finds a representative that is both correct (t and $rep_F(t)$ compare identically to times greater than F) and optimal (two times comparing identically to all times greater than F map to the same representative).

The formal properties of rep_F rely on properties of the $\wedge$ and $\vee$ operators, that they are respectively upper and lower bounds of their arguments, and their optimality:

$$b \leqslant a \text{ and } c \leqslant a \;\rightarrow\; (b \wedge c) \leqslant a \qquad (lub)$$
$$a \leqslant b \text{ and } a \leqslant c \;\rightarrow\; a \leqslant (b \vee c) \qquad (glb)$$

In particular, we will repeatedly use that if $t_1 \leqslant (t_2 \vee f)$ for all $f \in F$, then $t_1 \leqslant rep_F(t_2)$.

Theorem 1 (Correctness) *For any lattice element t and set F of lattice elements, $t \equiv_F rep_F(t)$.*

Proof 1 *We prove both directions of the implication in $\equiv_F$ separately, for all $f \geqslant F$. First assume $t \leqslant f$. By assumption, f is greater than some element f' of F, and so $t \wedge f' \leqslant f$ by the (lub) property. As a lower bound, $rep_F(t) \leqslant t \wedge f'$ for each $f' \in F$, and by transitivity $rep_F(t) \leqslant f$. Second assume $rep_F(t) \leqslant f$. Because $t \leqslant (t \wedge f')$ for all $f' \in F$, then $t \leqslant rep_F(t)$ by the (glb) property and by transitivity $t \leqslant f$.*

At the same time, rep_F is optimal in that two equivalent elements will be mapped to the same representative.

Theorem 2 (Optimality) *For any lattice elements t_1 and t_2 and set F of lattice elements, if $t_1 \equiv_F t_2$ then $rep_F(t_1) = rep_F(t_2)$.*

Proof 2 *For all $f \in F$ we have both that $t_1 \leqslant t_1 \wedge f$ and $f \leqslant t_1 \wedge f$, the latter implying that $t_1 \wedge f \geqslant F$. By our assumption, t_2 agrees with t_1 on times greater than F, making $t_2 \leqslant t_1 \wedge f$ for all $f \in F$. By correctness, $rep_F(t_2)$ agrees with t_2 on times greater than F, which includes $t_1 \wedge f$ for $f \in F$ and so $rep_F(t_2) \leqslant t_1 \wedge f$ for all $f \in F$. Because $rep_F(t_2)$ is less or equal to each term in the greatest lower bound definition of $rep_F(t_1)$, it is less or equal to $rep_F(t_1)$ itself. The symmetric argument proves that $rep_F(t_1) \leqslant rep_F(t_2)$, which implies that the two are equal (by antisymmetry).*

6.7 Evaluation

We evaluate DD on end-to-end workloads to measure the impact of shared arrangements with regards to query installation latency, throughput, and memory use (subsection 6.7.1). We then use microbenchmarks with DD to characterize our design's performance and the arrangement-aware operator implementations (subsection 6.7.2). Finally, we evaluate DD on pre-existing benchmarks across multiple domains to check if DD maintains high performance compared to other peer systems with and without using shared arrangements (subsection 6.7.3).

Implementation We implemented shared arrangements as part of DD, our stream processor. DD is our reference Rust implementation of Differential Dataflow [MD] with shared arrangements. It consists of a total of about 11,700 lines of code, and builds on an open-source implementation of Timely Dataflow [MT].

The `arrange` operator is defined in terms of a generic trace type, and our amortized merging trace is defined in terms of a generic batch type. Rust's static typing ensure that developers cannot incorrectly mix ordinary update triples and streams of arranged batches.

Setup We evaluate DD on a four-socket NUMA system with four Intel Xeon E5-4650 v2 CPUs, each with 10 physical cores and 512 GB of aggregate system memory. We compiled DD with `rustc` 1.33.0 and the

`jemalloc` [Jem] allocator. DD does distribute across multiple machines and supports sharding shared arrangements across them, but our evaluation here is restricted to multiprocessors. When we compare against other systems, we rely on the best, tuned measurements reported by their authors, but compare DD only if we are executing it on comparable or less powerful hardware than the other systems had access to.

6.7.1 End-to-end impact of shared arrangements

We start with an evaluation of shared arrangements in DD, in two domains with interactively issued queries against incrementally updated data sources. We evaluate the previously described streaming TPC-H setup, which windows the `lineitem` relation, as well as a recent interactive graph analytics benchmark. For the relational queries, we would hope to see shared arrangements reduce the installation latency and memory footprint of new queries when compared to an instance of DD that processes queries independently. For the graph tasks, we would hope that shared arrangements reduce the update and query latencies at each offered update rate, increase the peak update rate, and reduce the memory footprint when compared to an instance of DD that processes queries indepedently. In both cases, if shared arrangements work as designed, they should increase the capacity of DD on fixed resources, reducing the incremental costs of new queries.

6.7.1.1 TPC-H

The TPC-H [TPC] benchmark schema has eight relations, which describe order fulfillment events, as well as the orders, parts, customers, and suppliers they involve, and the nations and regions in which these entities exist. Of the eight relations, seven have meaningful primary keys, and are immediately suitable for arrangement (by their primary key). The eighth relation is `lineitem`, which contains fulfillment events, and we treat this collection as a stream of instantaneous events and do not arrange it.

TPC-H contains 22 "data warehousing" queries, meant to be run against large, static datasets. We consider a modified setup where the eight relations are progressively loaded [NDK16], one record at a time, in a round-robin fashion among the relations (at scale factor 10).

We focus on shared arrangements here, but DD matches or outperforms DBToaster [NDK16] even when queries run in isolation We show these results in subsubsection 6.7.3.4. To benchmark the impact of shared arrangements, we interactively deploy and retire queries while we load the eight relations. Each query has access to the full current contents of the seven keyed relations that we maintain shared arrangements for. By contrast, fulfillment events are windowed and each query only observes the fulfillment events from when it is deployed until when it is retired, implementing a "streaming" rather than a "historic" query. This evaluates the scenario presented in section 6.1, where analysts interactively submit queries. We report performance for ten active queries.

The 22 TPC-H queries differ, but broadly either derive from the windowed `lineitem` relation and reflect only current fulfillments, or they do not derive from `lineitem` and reflect the full accumulated volume of other relations. Without shared arrangements, either type of query requires building new indexed state for the seven non-`lineitem` relations. With shared indexes, we expect queries of the first type to be quick to deploy, as their outputs are initially empty. Queries of the second type should take longer to deploy in either case, as their initial output depends on many records.

Query latency To evaluate query latency, we measure the time from the start of query deployment until the initial result is ready to be returned. Query latency is significant because it determines whether the system delivers an interactive experience to human users, but also to dashboards that programmatically issue queries.

Figure 6.1a (shown in section 6.1) reports the distribution of query installation latencies, with and without shared arrangements. With shared arrangements, most queries (those that derive from `lineitem`) deploy and begin updating in milliseconds; the five queries that do not derive from `lineitem` are not windowed and perform non-trivial computation to produce their initial correct answer: they take between 100ms and 1s, depending on the sizes of the relations they use. Without shared arrangements, almost all queries take 1–2 seconds to install as they must create a reindexed copy of their inputs. Q1 and Q6 are exceptions, since they use no relations other than `lineitem`, and thus avoid reindexing any inputs; shared arrangements cannot improve the installation latency of

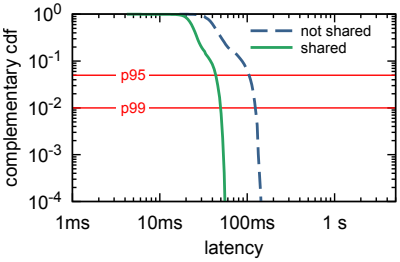
these queries. We conclude that shared arrangements substantially reduce the majority of query installation latencies, often by several orders of magnitude. The improvement to millisecond latency brings responses within interactive timescales, which helps improve productivity of human analysts and intervential latency for dependent software.

Update latency Once a query is installed, DD continually updates its results as new `lineitem` records arrive. To evaluate the update latency achieved, we record the amount of time required to process each round of input data updates after query installation.

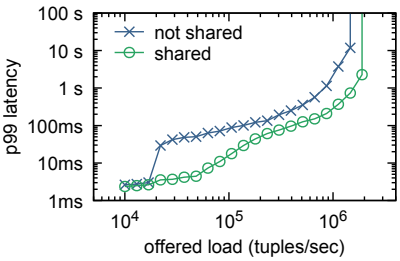
Figure 6.1b presents the distribution of these times, with and without shared arrangements, as a complementary cumulative distribution function (CCDF). The CCDF visualization—which we will use repeatedly—shows the "fraction of times with latency greater than" and highlights the tail latencies towards the bottom-right side of the plot. We see a modest but consistent reduction in processing time (about 2×) when using shared arrangements, which eliminate redundant index maintenance work. There is a noticeable tail in both cases, owed to two expensive queries that involve inequality joins (Q11 and Q22) and which respond slowly to changes in their inputs independently of shared arrangements. Shared arrangements yield lower latencies and increase update throughput.

Memory footprint Since shared arrangements eliminate duplicate copies of index structures, we would expect them to reduce the dataflow's memory footprint. To evaluate the memory footprint, we record the resident set size (RSS) as the experiment proceeds.

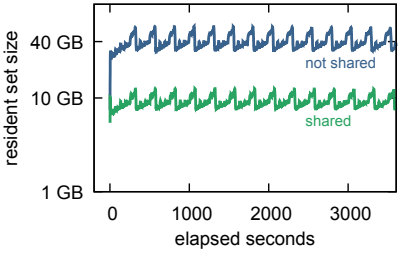
Figure 6.1c presents the timelines of the RSS with and without shared arrangements, and shows a substantial reduction (2–3×) in memory footprint when shared arrangements are present. Without shared arrangements, the memory footprint also varies substantially (between 60 and 120 GB) as the system creates and destroys indexes for queries that arrive and depart, while shared arrangements remain below 40 GB. Consequently, with shared arrangements, a given amount of system memory should allow for more active queries. In this experiment, ten concurrent queries are installed; workloads with more concurrent queries may have more sharing opportunities and achieve further memory economies.



(a) Latencies for query mix.



(b) 99^{th} percentile latency at given offered load.



(c) Resident set size.

Figure 6.4: Shared arrangements reduce query latency, increase the load handled, and reduce the memory footprint of interactive graph queries. The setup uses 32 workers, and issues 100k updates/sec and 100k queries/sec against a 10M node/64M edge graph in (a) and (c), while (b) varies the load. Note the $\log_{10}$–$\log_{10}$ scales in (a) and (b), and the $\log_{10}$-scale y-axis in (c).

6.7.1.2 Interactive graph queries

We further evaluate DD with an open-loop experiment issuing queries against an evolving graph. This experiment issues the four queries used by Pacaci et al. [Pac+17] to compare relational and graph databases: point look-ups, 1-hop look-ups, 2-hop look-ups, and 4-hop shortest path queries (shortest paths of length at most four). In the first three cases, the query argument is a graph node identifier, and in the fourth case it is a pair of identifiers.

We implement each of these queries as Differential Dataflows where the query arguments are independent collections that may be modified to introduce or remove specific query arguments. This query transformation was introduced in NiagaraCQ [Che+00] and is common in stream processors, allowing them to treat queries as a streaming input. The transformation can be applied to any queries presented as prepared statements. The dataflows depend on two arrangements of the graph edges, by source and by target; they are the only shared state among the queries.

We use a graph with 10M nodes and 64M edges, and update the graph and query arguments of interest at experiment-specific rates. Each graph update is the addition or removal of a random graph edge, and each query update is the addition or removal of a random query argument (queries are maintained while installed, rather than issued only once). All experiments evenly divide the query updates between the four query types.

Query latency We run an experiment with a fixed rate of 100,000 query updates per second, independently of how quickly DD responds to them. We would hope that DD responds quickly, and that shared arrangements of the graph structure should help reduce the latency of query updates, as DD must apply changes to one shared index rather than several independent ones.

Figure 6.4a reports the latency distributions with and without a shared arrangement of the graph structure, as a complementary CDF. Sharing the graph structure results in a 2–3× reduction in overall latency in the 95^{th} and 99^{th} percentile tail latency (from about 150ms to about 50ms). In both cases, there is a consistent baseline latency, proportional to the number of query classes maintained. Shared arrangements yield latency reductions across all query classes, rather than, e.g. imposing the latency

System	#	look-up	one-hop	two-hop	4-path
Neo4j	32	9.08ms	12.82ms	368ms	21ms
Postgres	32	0.25ms	1.4ms	29ms	2242ms
Virtuoso	32	0.35ms	1.23ms	11.55ms	4.81ms
DD, 10^0	32	0.64ms	0.92ms	1.28ms	1.89ms
DD, 10^1	32	0.81ms	1.19ms	1.65ms	2.79ms
DD, 10^2	32	1.26ms	1.79ms	2.92ms	8.01ms
DD, 10^3	32	5.71ms	6.88ms	10.14ms	72.20ms

Table 6.2: On comparable 10M node/64M edge graphs, DD is broadly competitive with the average graph query latencies of three systems evaluated by Pacaci et al. [Pac+17], and scales to higher throughput using batching. The DD batch size is the number of concurrent queries per measurement.

of the slowest query on all sharing dataflows. This validates that queries can proceed at different rates, an important property of our shared arrangement design.

Update throughput To test how DD's shared arrangements scale with load, we next scale the rates of graph updates and query changes up to two million changes per second each. An ideal result would show that sharing the arranged graph structure consistently reduces the computation required, thus allowing us to scale to a higher load using fixed resources.

Figure 6.4b reports the 99^{th} percentile latency with and without a shared graph arrangement, as a function of offered load and on a log–log scale. The shared configuration results in reduced latencies at each offered load, and tolerates an increased maximum load at any target latency. At the point of saturating the server resources, shared arrangements tolerate 33% more load than the unshared setup, although this number is much larger for specific latencies (e.g. 5× at a 20ms target). We note that the absolute throughputs achieved in this experiment exceed the best throughput observed by Pacaci et al. (Postgres, at 2,000 updates per second) by several orders of magnitude, further illustrating the benefits of parallel dataflow computation with shared arrangements.

Memory footprint Finally, we consider the memory footprint of the computation. There are five uses of the graph across the four queries, but

also per-query state that is unshared, so we would expect a reduction in memory footprint of somewhat below 4×.

Figure 6.4c reports the memory footprint for the query mix with and without sharing, for an hour-long execution. The memory footprint oscillates around 10 GB with shared arrangements, and around 40 GB (4× larger) without shared arrangements. This illustrates that sharing state affords memory savings proportional to the number of reuses of a collection.

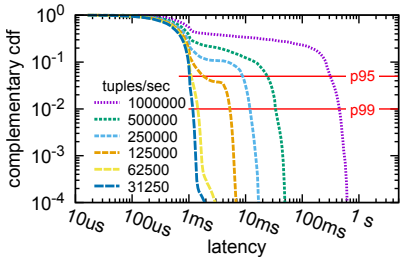
6.7.1.3 Comparison with other systems

Pacaci et al. [Pac+17] evaluated relational and graph databases on the same graph queries. DD is a stream processor rather than a database and supports somewhat different features, but its performance ought to be comparable to the databases' for these queries. We stress, however, that our implementation of the queries as Differential Dataflows requires that queries be expressed as prepared statements, a restriction the other systems do not impose.

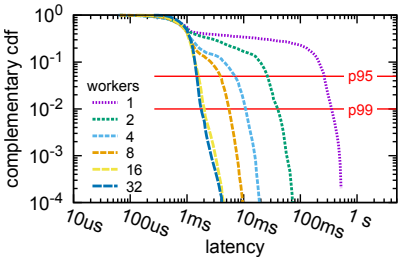
We ran DD experiments with a random graph comparable to the one used in Pacaci et al.'s comparison. Table 6.2 reports the average latency to perform and then await a single query in different systems, as well as the time to perform and await batches of increasing numbers of concurrent queries for DD. While DD does not provide the lowest latency for point look-ups, it does provides excellent latencies for other queries and increases query throughput with batch size.

6.7.2 Design evaluation

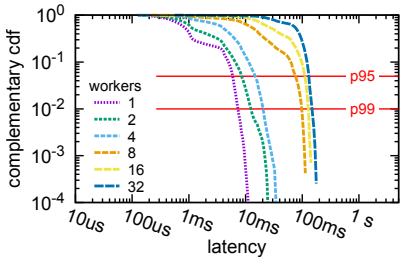
We now perform microbenchmarks of the `arrange` operator, to evaluate its response to changes in load and resources. In all benchmarks, we apply an `arrange` operator to a continually changing collection of 64-bit identifiers (with 64-bit timestamp and signed difference). The inputs are generated randomly at the worker, and exchanged (shuffled) by key prior to entering the arrangement. We are primarily interested in the distribution of response latencies, as slow edge-case behavior of an arrangement would affect this statistic most. We report all latencies as complementary CDFs to get high resolution in the tail of the distribution.



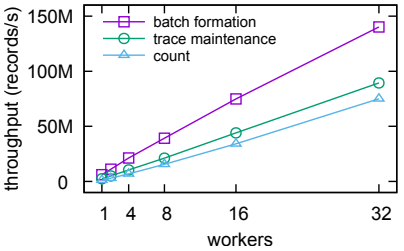
(a) Varying offered load with 1 worker.



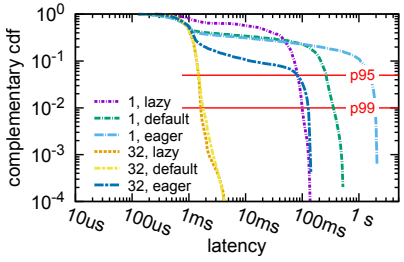
(b) Varying workers with fixed load.



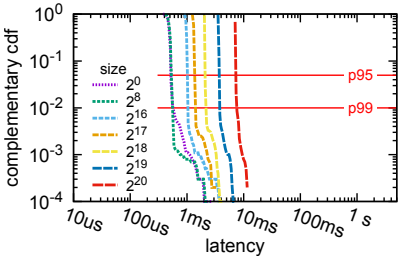
(c) Varying workers and offered load



(d) Task throughput, varying workers.



(e) Amortized merging levels.



(f) Join with pre-arranged collection.

Figure 6.5: Microbenchmarks of our shared arrangement design suggest that our design scales well with growing parallelism (b)–(d) and load (a), (c)–(d), and that the key ideas of amortized merging (e) and proportional work across inputs (f) are crucial to achieving low update latencies. (b) and (e) generate a fixed load of 1M input records per second.

Varying load As update load varies, our shared arrangement design should trade latency for throughput until equilibrium is reached. Figure 6.5a reports the latency distributions for a single worker as we vary the number of keys and offered load in an open-loop harness, from 10M keys and 1M updates per second, downward by factors of two. Latencies drop as load decreases, down to the test harness's limit of one millisecond. This demonstrates that arrangements are suitable for both low-latency and high-throughput.

Strong scaling More parallel workers should allow faster maintenance of a shared arrangement, as the work to update it parallelizes, unless coordination frequency interferes. Figure 6.5b reports the latency distributions for an increasing numbers of workers under a fixed load of 10M keys and 1M updates per second. As the number of workers increases, latencies decrease, especially in the tail of the distribution: for example, the 99^{th} percentile latency of 500ms with one worker drops to 6ms with eight workers.

Weak scaling Varying the number of workers while proportionately increasing the number of keys and offered load would ideally result in constant latency. Figure 6.5c shows that the latency distributions do exhibit increased tail latency, as the act of data exchange at the arrangement input becomes more complex. However, the latencies do stabilize at 100–200ms as the number of workers and data increase proportionately.

Throughput scaling An arrangement consists of several subcomponents: batch formation, trace maintenance, and e.g. a maintained `count` operator. To evaluate throughput scaling, we issue batches of 10,000 updates at each worker, repeated as soon as each batch is accepted, rather than from a rate-limited open-loop harness. Figure 6.5d reports the peak throughputs as the number of cores (and thus, workers and arrangement shards) grows. All components scale linearly to 32 workers.

Amortized merging The amortized merging strategy is crucial for shared arrangements to achieve low update latency, but its efficacy depends on setting the right amortization coefficients. Eager merging performs the least work overall but can increase tail latency. Lazy merging

performs more work overall, but should reduce the tail latency. Ideally, DD's default would pick a good tradeoff between common-case and tail latencies at different scales.

Figure 6.5e reports the latency distributions for one and 32 workers, each with three different merge amortization coefficients: the most eager, DD's default, and the most lazy possible. For a single worker, lazier settings have smaller tail latencies, but are more often in that tail. For 32 workers, the lazier settings are significantly better, because eager strategies often cause workers to stall waiting for a long merge at one worker. The lazier settings are critical for effective strong scaling, where eager work causes multiple workers to seize up, which matches similar observations about garbage collection at scale [Gog+15]. DD's default setting achieves good performance at both scales.

Join proportionality Our arrangement-aware join operator is designed to perform work proportional to the size of the smaller of the incoming pre-arranged batch and the state joined against (subsubsection 6.5.3.1). We validate this by measuring the latency distributions to install, execute, and complete new dataflows that join collections of varying size against a pre-existing arrangement of 10M keys.

The varying lines in Figure 6.5f demonstrate that the join work is indeed proportional to the small collection's size, rather than to the (constant) 10M arranged keys. This behavior is not possible in a record-at-a-time stream processor, which must at least examine each input record. This behavior is possible in DD only because the `join` operator receives as input pre-arranged batches of updates. Query deployment in the TPC-H workload would not be fast without this property.

6.7.3 Baseline performance on reference tasks

We also evaluate DD against established prior work to demonstrate that DD is competitive with and occasionally better than peer systems. Importantly, these established benchmarks are traditionally evaluated in isolation, and are rarely able to demonstrate the benefits of shared arrangements. Instead, this evaluation is primarily to demonstrate that DD does not *lose* baseline performance as compared to other state-of-the-art systems. Most but not all of the peer systems in this section do maintain

private indexed data in operators; this decision alone accounts for some of the gaps.

6.7.3.1 Datalog workloads

Datalog is a relational language in which queries are sets of recursively defined productions, which are iterated from a base set of records until no new records are produced. Unlike graph computation, Datalog queries tend to produce and work with substantially more records than they are provided as input. Several shared-memory systems for Datalog exist, including LogicBlox, DLV [DLV], DeALS [YSZ17], and several distributed systems have recently emerged, including Myria [Wan+17a], SociaLite [SGL15], and BigDatalog [Shk+16]. At the time of writing, only LogicBlox supports decremental updates to Datalog queries, using a technique called "transaction repair" [Vel14]. DD supports incremental and decremental updates to Datalog computations and interactive top-down queries.

Top-down (interactive) evaluation Datalog users commonly specify values in a query, such as *reach*(*"david"*, ?), to request nodes reachable from a source node. The 'magic set' transformation [Ban+86] rewrites such queries as bottom-up computations with a new base relation that seeds the bottom-up derivation with query arguments; the rewritten rules derive facts only with the participation of some seed record. DD, like some interactive Datalog environments, performs this work against maintained arrangements of the non-seed relations. We would expect this approach to be much faster than full evaluation, which batch processors that re-index the non-seed relations (or DD without shared arrangements) require.

Table 6.3 reports DD's median and maximum latencies for 100 random arguments for three interactive queries on three widely-used benchmark graphs, and the times for full evaluation of the related query, using 32 workers. DD's arrangements mostly reduce runtimes from seconds to milliseconds. The slower performance for sg(x,?) on grid-150 reveals that the transformation is not always beneficial, a known problem with the magic set transform.

Query	statistic	tree-11	grid-150	gnp1
tc(x,?)	increm., median	2.56ms	346.28ms	18.29ms
	incremental, max	9.05ms	552.79ms	25.40ms
	full eval. (no SA)	0.08s	6.18s	9.45s
tc(?,x)	increm., median	15.63ms	320.83ms	15.58ms
	incremental, max	18.01ms	541.76ms	23.84ms
	full evaluation	0.08s	6.18s	9.45s
sg(x,?)	increm., median	68.34ms	1075.11ms	20.08ms
	incremental, max	95.66ms	2285.11ms	26.56ms
	full eval. (no SA)	56.45s	0.60s	19.85s

Table 6.3: DD enables interactive computation of three Datalog queries (32 workers, medians and maximums over 100 queries). Full evaluation is required without shared arrangements.

System	cores	tc(t)	tc(g)	tc(r)	sg(t)	sg(g)	sg(r)
BigDatalog	120	49s	25s	7s	53s	34s	72s
Spark	120	244s	OOM	63s	OOM	1955s	430s
Myria	120	91s	22s	50s	822s	5s	436s
SociaLite	120	DNF	465s	654s	OOM	17s	OOM
LogicBlox	64	NR	24420s	913s	58732s	326s	3363s
DLV	1	NR	13127s	9272s	OOM	105s	48039s
DeALS	1	NR	148s	321s	1309s	7.6s	2926s
DeALS	64	NR	5s	12s	48s	0.35s	79s
DD	1	98.26s	132.23s	210.25s	1210.78s	4.43s	482.91s
DD	2	53.42s	68.13s	111.98s	652.74s	2.76s	253.80s
DD	4	27.85s	34.42s	57.69s	325.24s	1.63s	125.00s
DD	8	15.37s	17.97s	30.90s	173.96s	1.06s	66.10s
DD	16	9.63s	9.74s	16.66s	93.47s	0.69s	35.44s
DD	32	7.18s	6.18s	9.45s	56.45s	0.60s	19.85s

Table 6.4: System performance on various Datalog problems and graphs. Times for the first four systems are reproduced from [Shk+16]. NR indicates the measurements were not reported, DNF indicates a run that lasted more than 24 hours, and OOM indicates the system terminated due to lack of memory.

Bottom-up (batch) evaluation Table 6.4 reports elapsed seconds first for distributed systems, then for single-machine systems, and then for DD at varying numbers of workers. The workloads are "transitive closure" (*tc*) and "same generation" (*sg*) on supplied graphs that are trees (*t*), grids (*g*), and random graphs (*r*).

DD is generally competitive with the best of the specialized Datalog systems (here: DeALS), and generally out-performs the distributed data processors. BigDatalog competes well on transitive closure due to an optimization for linear queries where it broadcasts its input dataset to all workers; this works well with small inputs, as here, but is not generally a robust strategy.

6.7.3.2 Program Analysis

Graspan [Wan+17b] is a system built for static analysis of large code bases, created in part because existing systems were unable to handle non-trivial analyses at the sizes required. Wang et al. benchmarked Graspan for two program analyses, *dataflow* and *points-to* [Wan+17b]. The *dataflow* query propagates null assignments along program assignment edges, while the more complicated *points-to* analysis develops a mutually recursive graph of value flows, and memory and value aliasing. We developed a full implementation of Graspan—query parsing, dataflow construction, input parsing and loading, dataflow execution—in 179 lines of code on top of DD.

Graspan is designed to operate out-of-core, and explicitly manages its data on disk. We therefore report DD measurements from a laptop with only 16 GB of RAM, a limit exceeded by the *points-to* analysis (which peaks around 30 GB). The sequential access in this analysis makes standard OS swapping mechanisms sufficient for out-of-core execution, however. To verify this, we modify the computation to use 32-bit integers, reducing the memory footprint below the RAM size, and find that this optimized version runs only about 20% faster than the out-of-core execution.

Table 6.5a and Table 6.6a show the running times reported by Wang et al. compared to those DD achieves. For both queries, we see a substantial improvement (from 24× to 650×). The *points-to* analysis is dominated by the determination of a large relation (value aliasing) that is used only once. This relation can be optimized out, as value aliasing is eventually

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	4 hrs
Graspan	4	713.8 min	143.8 min	11.3 min
RecStep	20	430s	359s	74s
DD	1	65.8s	32.0s	8.9s

(a) *dataflow* query, DD on laptop hardware.

System	cores	*linux*	*psql*	*httpd*
RecStep	20	430s	359s	74s
DD	2	53.9s	25.5s	7.5s
DD	4	34.8s	16.3s	4.7s
DD	8	24.4s	11.2s	3.2s
DD	16	20.7s	8.7s	2.5s

(b) *dataflow* query, DD on server hardware.

System	cores	*linux (kernel only)*	*psql*	*httpd*
DD (med)	1	1.05ms	143ms	18.1ms
DD (max)	1	7.34ms	1.21s	201ms

(c) Times to remove each of the first 1,000 null assignments from the interactive top-down *dataflow* query.

Table 6.5: DD performs well for Graspan [Wan+17b] *dataflow* query on three graphs. SociaLite and Graspan results from Wang et al. [Wan+17b]; RecStep results from Fan et al. [Fan+19]; OOM: out of memory.

System	cores	*linux*	*psql*	*httpd*
SociaLite	4	OOM	OOM	> 24 hrs
Graspan	4	99.7 min	353.1 min	479.9 min
RecStep	20	61s	162s	162s
DD	1	241.0s	151.2s	185.6s
DD (Opt)	1	121.1s	52.3s	51.8s

(a) *points-to* analysis, DD on laptop. DD (Opt) is an optimized query.

System	cores	*linux*	*psql*	*httpd*
RecStep	20	61s	162s	162s
DD	2	230.0s	134.4s	145.3s
DD	4	142.6s	73.3s	80.2s
DD	8	86.0s	40.9s	44.9s
DD	16	59.8s	24.0s	27.5s
DD (Opt)	2	125.2ss	53.1ss	46.0s
DD (Opt)	4	89.8s	30.8s	26.7s
DD (Opt)	8	57.4s	18.0s	15.1s
DD (Opt)	16	43.1s	11.2s	9.1s

(b) *points-to* analysis, DD on server. DD (Opt) is an optimized query.

Table 6.6: DD performs well for Graspan [Wan+17b] program analyses on three graphs. SociaLite and Graspan results from Wang et al. [Wan+17b]; RecStep results from Fan et al. [Fan+19]; OOM: out of memory.

restricted by dereferences, and this restriction can be performed before forming all value aliases. This optimization results in a more efficient computation, but one that reuses some relations several (five) times; the benefits of the improved plan may not be realized by systems without shared arrangements. Table 6.6a reports the optimized running times as (Opt).

In Table 6.5b and Table 6.6b we also report the runtimes of DD on these program analysis tasks on server hardware (with the same hardware configuration as previous sections) and compare them to RecStep [Fan+19], a state-of-the-art parallel datalog engine. For all queries, DD matches or outperforms RecStep running times even when it is configured to utilize a smaller number of CPU cores.

Top-down evaluation Both *dataflow* and *points-to* can be transformed to support interactive queries instead of batch computation. Table 6.5c reports the median and maximum latencies to remove the first 1,000 null assignments from the completed *dataflow* analysis and correct the set of reached program locations. While there is some variability, the timescales are largely interactive and suggest the potential for an improved developer experience.

6.7.3.3 Batch graph computation

We evaluate DD on standard batch iterative graph computations on three standard social networks: LiveJournal, Orkut, and Twitter.

Following prior work [Shk+16] we use the tasks of single-source reachability (reach), single-source shortest paths (sssp), and undirected connectivity (wcc). For the first two problems we start from the first graph vertex with any outgoing edges (each reaches a majority of the graph).

We separately report the times required to form the forward and reverse edge arrangements, with the former generally faster than the latter as the input graphs are sorted by the source as in the forward index. The first two problems require a forward index and undirected connectivity requires indices in both directions, and we split the results accordingly. We report times for three graphs: `livejournal` in Table 6.8, `orkut` in Table 6.9, and `twitter` in Table 6.7. We include measurements by Shkapsky et al. [Shk+16] for several other systems. We also reproduce

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	14.89s	14.89s	-	33.99s
w/hash map	1	-	192.01s	192.01s	-	404.19s
BigDatalog	120	-	125s	260s	-	307s
Myria	120	-	102s	1593s	-	1051s
SociaLite	120	-	755s	OOM	-	OOM
GraphX	120	-	3677s	6712s	-	12041s
RaSQL	120	-	45s	81s	-	108s
RecStep	20	-	174s	243s	-	501s
DD	1	162.41s	256.77s	310.63s	312.31s	800.05s
DD	2	99.74s	131.50s	159.93s	164.12s	417.20s
DD	4	49.46s	64.31s	77.27s	81.67s	200.28s
DD	8	27.99s	33.68s	40.24s	43.20s	101.42s
DD	16	18.04s	17.40s	20.99s	24.73s	51.83s
DD	32	12.69s	11.36s	10.97s	14.44s	27.48s

Table 6.7: System performance on various tasks on the 42M node, 1.4B edge `twitter` graph. DD does not share any arrangements here, but the sharing infrastructure does not harm performance.

measurements reported in [Shk+16] for several other systems. We include running times for simple single-threaded implementations that are not required to follow the same algorithms. For example, for undirected connectivity we use the union-find algorithm rather than label propagation, which outperforms all systems except DD at 32 cores. We also include the same times when the single-threaded implementations replace the arrays storing per-node state with hash maps, as they might when the graph identifiers have not be pre-processed to be in a compact range; the graphs remain densely packed and array indexed.

DD is consistently faster than the other systems—Myria [Wan+17a], BigDatalog [Shk+16], SociaLite [SGL15], GraphX [Gon+14], RecStep [Fan+19], and RaSQL [Gu+19]—but is substantially less efficient than purpose-written single-threaded code applied to pre-processed graph data. Such pre-processing is common, as it allows use of efficient static arrays, but it prohibits more general vertex identifiers or graph updates. When we amend our purpose-built code to use a hash table instead of an array, DD becomes competitive between two and four cores. These results are independent of shared arrangements, but indicate that DD's arrangement-aware implementation does not impose any undue cost on computations without sharing.

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	0.40s	0.40s	-	0.29s
w/hash map	1	-	4.38s	4.38s	-	8.90s
BigDatalog	120	-	17s	53s	-	27s
Myria	120	-	5s	70s	-	39s
SociaLite	120	-	52s	172s	-	54s
GraphX	120	-	36s	311s	-	59s
RecStep	20	-	14s	19s	-	39s
DD	1	4.39s	8.50s	13.14s	7.56s	23.97s
DD	2	2.49s	4.33s	6.71s	4.01s	12.95s
DD	4	1.39s	2.31s	3.58s	2.06s	6.29s
DD	8	0.74s	1.20s	1.79s	1.03s	3.41s
DD	16	0.54s	0.62s	0.90s	0.58s	1.71s
DD	32	0.55s	0.51s	0.59s	0.41s	0.90s

Table 6.8: System performance on various tasks on the 4.8M node, 68M edge `livejournal` graph.

System	cores	index-f	reach	sssp	index-r	wcc
Single thread	1	-	0.46s	0.46s	-	0.52s
w/hash map	1	-	11.56s	11.56s	-	19.00s
BigDatalog	120	-	20s	39s	-	33s
Myria	120	-	6s	44s	-	57s
SociaLite	120	-	67s	106s	-	78s
GraphX	120	-	48s	67s	-	53s
RaSQL	120	-	11s	16s	-	19s
RecStep	20	-	19s	25s	-	43s
DD	1	14.02s	20.33s	24.65s	21.27s	47.79s
DD	2	7.92s	10.29s	13.06s	11.49s	25.02s
DD	4	4.25s	5.34s	6.21s	5.73s	12.38s
DD	8	2.37s	2.68s	3.34s	3.03s	6.29s
DD	16	1.43s	1.47s	1.60s	1.69s	3.30s
DD	32	1.22s	1.11s	0.87s	1.05s	1.75s

Table 6.9: System performance on various tasks on the 3M node, 117M edge `orkut` graph.

query	DBToaster	DD (w=1)	DD (w=32)
TPC-H 01	4,372,480	9,341,713	31,283,993
TPC-H 02	691,260	4,388,761	29,651,632
TPC-H 03	4,580,003	11,049,606	37,263,673
TPC-H 04	9,752,380	9,046,854	30,886,269
TPC-H 05	509,490	5,802,513	27,952,246
TPC-H 06	101,327,791	33,090,863	65,335,474
TPC-H 07	646,018	7,551,628	30,962,626
TPC-H 08	221,020	4,949,412	28,230,062
TPC-H 09	76,296	2,932,421	18,119,469
TPC-H 10	5,964,290	9,708,371	25,037,510
TPC-H 11	591,716	1,720,655	1,749,464
TPC-H 12	7,469,474	11,258,702	33,975,983
TPC-H 13	474,765	1,446,223	16,792,703
TPC-H 14	53,436,252	21,908,762	38,843,085
TPC-H 15	964	5,057,397	23,122,916
TPC-H 16	58,721	4,435,818	23,495,608
TPC-H 17	131,964	5,218,907	25,888,103
TPC-H 18	971,313	5,854,293	29,574,347
TPC-H 19	8,776,165	22,696,357	36,393,109
TPC-H 20	1,871,407	16,089,949	46,456,453
TPC-H 21	407,540	1,968,771	10,928,516
TPC-H 22	815,903	1,843,397	15,233,935

Table 6.10: Streaming update rates (in tuples per second) for the 22 TPC-H queries at scale factor 10, with logical batching of 100,000 elements at the same time. Elapsed times for DBToaster are for one thread, and are reproduced from [NDK16]. For DD we report both one worker and 32 worker rates.

6.7.3.4 Relational computations

Table 6.10 reports throughput in tuples per second for [NDK16] and DD on the scale factor 10 TPC-H workload with logical batches of 100,000 elements. DD has a generally higher and more consistent rate, though is less efficient on lighter queries (q04, q06, and q14); DD performs no pre-aggregation, which could improve its rates for logically batched queries. For 32 workers, almost all rates are above 10 million updates per second, which correspond to latencies below 10ms between reports.

Table 6.11 reports elapsed times for DD applied to the scale-factor 10 TPC-H workload. We also reproduce several measurements from [Ess+18] for other systems. All are executed with a single thread. DD used as a batch processor is not as fast as the best systems (HyPer and Flare), but is faster than other popular frameworks.

query	Postgres	SparkSQL	HyPer	Flare	DD
TPC-H 01	241,404	18,219	603	530	7,789
TPC-H 02	6,649	23,741	59	139	2,426
TPC-H 03	33,721	47,816	1,126	532	5,948
TPC-H 04	7,936	22,630	842	521	8,550
TPC-H 05	30,043	51,731	941	748	14,001
TPC-H 06	23,358	3,383	232	198	1,185
TPC-H 07	32,501	31,770	943	830	12,029
TPC-H 08	29,759	63,823	616	1,525	19,667
TPC-H 09	64,224	88,861	1,984	3,124	27,873
TPC-H 10	33,145	42,216	967	1,436	4,559
TPC-H 11	7,093	3,857	131	56	1,534
TPC-H 12	37,880	17,233	501	656	4,458
TPC-H 13	31,242	28,489	3,625	3,727	3,893
TPC-H 14	22,058	7,403	330	278	1,695
TPC-H 15	23,133	14,542	253	302	1,591
TPC-H 16	13,232	23,371	1,399	620	2,238
TPC-H 17	155,449	70,944	563	2,343	17,750
TPC-H 18	90,949	53,932	3,703	823	9,426
TPC-H 19	29,452	13,085	1,980	909	2,444
TPC-H 20	65,541	31,226	434	870	4,658
TPC-H 21	299,178	128,910	1,626	1,962	29,363
TPC-H 22	11,703	10,030	180	177	2,819

Table 6.11: Elapsed milliseconds for the 22 TPC-H queries at scale factor 10, each using a single core. Elapsed times for the four other systems are reproduced from [Ess+18]. DD used as a batch processor is not as fast as the best systems (HyPer and Flare), but is faster than other popular frameworks.

6.8 Conclusions

We described shared arrangements, detailed their design and implementation in DD, and showed how they yield improved performance for interactive analytics against evolving data. Shared arrangements enable interactive, incrementally maintained queries against streams by sharing sharded indexed state between operators within or across dataflows. Multiversioning the shared arrangement is crucial to provide high throughput, and sharding the arrangement achieves parallel speedup. Our implementation in DD installs new queries against a stream in milliseconds, reduces the processing and space cost of multiple dataflows, and achieves high performance on a range of workloads. In particular, we showed that shared arrangements improve performance for workloads with concurrent queries, such as a streaming TPC-H workload with interactive analytic queries and concurrent graph queries.

Shared arrangements rely on features shared by time-aware dataflow systems, and the idiom of a single-writer, multiple-reader index should apply to several other popular dataflow systems. We left undiscussed topics like persistence and availability. As a deterministic data processor, DD is well-suited to active-active replication for availability in the case of failures. In addition, the immutable LSM layers backing arrangements are appropriate for persistence, and because of their inherent multiversioning can be persisted asynchronously, off of the critical path. We present a techinque for persistence and fault tolerance of time-aware dataflow systems that utilizes the trace data structure in Chapter 7.

7 Fault tolerance

Parts of this chapter describe joint work with Lorenzo Selvatici in the context of his Master Thesis "A Streaming System with Coordination-Free Fault-Tolerance" [Sel04]; an early prototype of the mechanism described in this chapter was primarily designed and implemented by me with some contribution by Frank McSherry and Moritz Hoffmann.

A long-running dataflow program accumulates in-memory state in the operators. This state acts as a summary of the data processed at each operator so far, it is critical for producing future results, and – depending on the computation – may require the entire input history for an operator to be faithfully reconstructed. For example, an operator computing a running total summarizes its entire input history as its internal state (the total).

When a failure occurs, either due to hardware or a software bug, the in-memory state of the operator instances residing on the failing nodes is lost. A fault-tolerance mechanism enables persisting and recovering such state so that the time-aware dataflow program can quickly resume producing outputs. If such a mechanism isn't present, resuming the program may require re-processing the entire input history, which can easily be prohibitively expensive.

Even if it's feasible to re-process the entire input history or only the input relevant to reconstruct the current state, a time-aware dataflow program may have variably strict latency requirements due to service-level-agreements in producing output in response to new inputs which cannot be met if the system needs to recompute the state from the inputs.

This chapter describes a new mechanism for preserving and recovering the internal, intermediate state of time-aware dataflow operator instances

which has no latency spikes in steady-state execution, and can recover quickly after failure to meet strict latency SLAs.

7.1 Introduction

Stream processing systems that process data continuously from an external data sources accumulate intermediate results as internal in-memory state. When part of these systems fail, either due to crash-bugs or hardware failures, this ephemeral in-memory state is lost. A durable copy of the source data stream can be prohibitively large, and even if there is one, re-processing the entire input to reconstruct the state is often too expensive or incompatible with the application latency requirements.

Dataflow streaming systems generally run as a distributed system over many cores or computers communicating via reliable channels, and the data is sharded across the available workers to enable parallelism. For correct recovery we need to acquire and recover from checkpoints that faithfully represent a global state of the system [CL85]. In a distributed system the workers do not share a clock, or memory, and thus they cannot record their local state at the same instant.

Prior work for durability in distributed dataflow systems devised protocols to coordinate when to take a snapshot on all workers in order to obtain a consistent representation of the global state. Taking complete global snapshots takes time, is expensive, and generally requires partially pausing the system to guarantee consistency. We explore these limitations in more detail in section 7.2.

We observe that modern time-aware [McS+20] stream processing systems all employ message timestamps that represent logical and causal relationships between data messages across the system. These timestamps are used for coordination, and allow operators to identify when they have received all input for a certain timestamp "epoch" so that they can correctly compute complete results.

We propose to leverage these existing logical consistency boundaries for durability and recovery in time-aware systems, instead of building an additional out-of-band coordination mechanism like in previous work. Our durability and recovery protocol for pipelined dataflow systems, CL, precisely records the history of each of the worker's state independently, relieving the need for coordination on the critical path; the protocol only

constructs a global consistent state after failure, by relying on causality cues given by message timestamps.

Figure 7.1 shows preliminary evaluation results for CL. It compares latency over time for a simple but representative workload (the widely used NexMark Q5) between our CL prototype and the baseline non-fault-tolerant timely dataflow. We offered a costant load at roughly 60% of the system's maximum throughtput when using CL and, for the CL experiment, injected failures at 200 and 400 seconds. We see that CL delivers predictable steady-state latency, albeit higher than the baseline non-fault-tolerant system. It achieves this by doing away with the coordination necessary to record snapshots that faithfully represent a global state. Additionally, thanks to the high granularity of the state recording, it recovers fairly quickly.

This predictable latency comes at some throughput cost due to the additional work involved in persisting the state history. Our preliminary results indicate that CL should be able to achieve at least roughly half the throughput of a non-fault-tolerant timely dataflow implementation. This means that using CL for fault-tolerance requires roughly the same resources as running active-active replication (where the resource requirements are doubled by running a second copy of the system). However active-active replication can only survive a single worker failiure, while CL can survive an arbitrary number, depending on the exact durability configuration.

We built CL as a library on top of the timely dataflow[MT] stream processing system using timestamp tokens to encode the coordination protocol used by CL to (i) asynchronously compact and garbage collect older recorded state and to (ii) construct globally consistent state on recovery depending on which state updates had been successfully persisted before failure. We will see the details of these protocols in section 7.7 and **??**.

7.2 Related work

Streaming engines based on **batch processors**, such as Spark Streaming [Zah+13], use the natural boundaries of fixed sized *micro-batches* to store and replicate state. The fixed-sized batches, however, impose higher

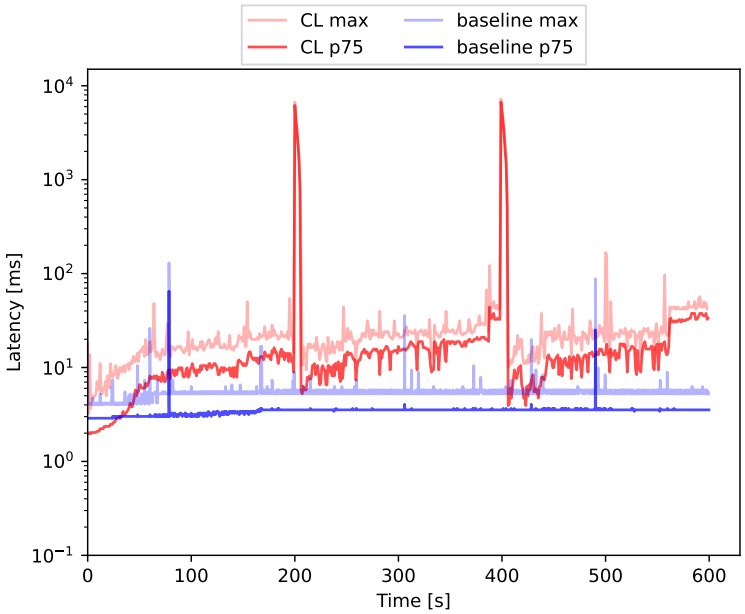


Figure 7.1: NexMark Q5 comparing latency of a non-fault-tolerant baseline with CL. The offered load was 5000 tuples/sec using a configuration with three timely dataflow workers.

minimum latency and afford reduced flexibility compared to pipelined streaming systems.

Systems like MillWheel rely on an **external consistent store** for its state, and handle the processing of incoming messages as transactions that are atomically committed when complete. The additional round-trip to the datastore adds latency, and its serialization constraints reduce the opportunity for pipelining.

Pipelined systems like Flink [Car+17] rely on **in-band markers** that travel along with the data and trigger local state snapshots at the operators they encounter. Aligning markers from multiple workers requires pausing processing, making it unfeasible to capture snapshot with high frequency.

7.3 Assumptions

Because our recovery process requires recomputing intermediate state that was loss on failure, we need to impose some restrictions on the operators in the dataflow program: in particular, they must

- deterministically update the state after they have received all inputs for a certain timestamp, regardless of the order in which those inputs have been received;
- produce semantically equivalent outputs for a certain timestamp regardless of the order in which the inputs have been received; and
- use a specific programming interface to perform state updates so that they can be made durable by the fault-tolerance mechanism.

Most time-aware dataflow operators already satisfy the first two restrictions, or can be straightforwardly modified to fit while retaining their semantics and with minimal performance impact. For example The last restriction requires a more significant refactor of existing time-aware dataflow operators.

7.4 Failure model

The durability and recovery protocol described in this chapter is partially agnostic to the specific deployment and configuration choice, which will

depend on the types and frequency of failures that the operator wants to safeguard against. The only requirement is that (i) when a worker fails, some mechanism can identify a replacement worker (potentially just a new process on the same machine, if the failure was just a crash); and (ii) the replacement worker has access to the state log of its predecessor for all items that had been acknowledged as successfully persisted.

The operator can choose the persistent storage and configuration depending on the durability and availability requirements of the application. CL's interface with the persistent storage is simple `get` and `set` operations that asynchronously persist and retrieve blobs of data associated with a key; this interface is typical of object storage systems in the cloud [Ama].

7.5 Consistency goal

Modern time-aware dataflow systems provide serializability guarantees for their outputs at timestamp boundaries: all the input data pertaining to a certain timestamp t, and no other data, must be reflected in the output as of timestamp t.

Given the constraints described in section 7.3, the goal is for the system that experiences a failure to produce outputs identical to what would have been produced if a failure did not happen, and thus to preserve the serializability guarantee in case of failure. This is sometimes referred to as "exactly-once" semantics : the output of the system should match that of an execution where each record is processed exactly one time, even when records may have to be re-processed during the recovery process.

We assume that the *inputs* of the computation are stored in a durable queue and recent inputs can be replayed on request of the system in case of failure. CL can provide a periodic signal to the durable input queue informing it of which prefix of the input can be safely garbage collected without compromising the ability of the computation to recover from a failure.

7.6 CL operators

Our model of a CL operator refines the time-aware dataflow operator model in order to satisfy the restrictions described in section 7.3. Fig-

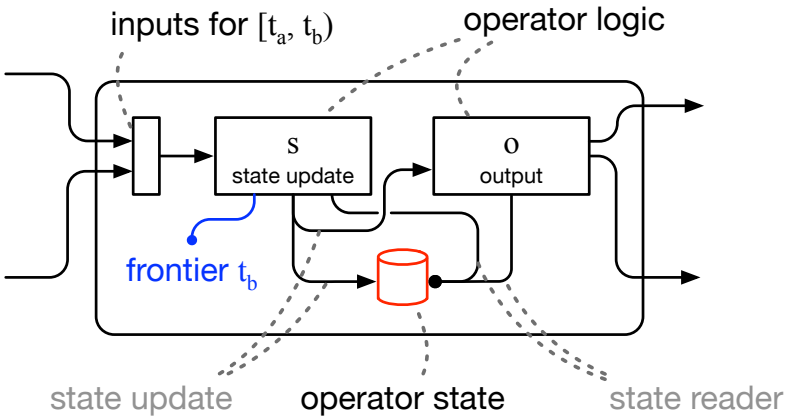


Figure 7.2: A CL operator.

ure 7.2 shows a diagram of a CL operator. An operator's behavior is defined by the programmer via two code fragments: a state update function s and an output generation function o.

The **state update function** s receives the inputs for a range of complete timestamps $[t_a, t_b)$, is informed of the frontier information for the input ports t_b, and has access to an immutable view of the operator state. Using this information s is tasked with computing the state update resulting from inputs with timestamps $[t_a, t_b)$: instead of directly mutating the operator state, s must produce a delta from the previous state for each intermediate timestamp between t_a and t_b.

The **output generation function** o receives the state update computed by s and is tasked with producing the corresponding output messages by also relying on an immutable view of the operator state.

The two functions s and o are sufficient to define the data transformation associated with the operator. Because the state update function s produces state deltas at each timestamp, the system is able to track and persist these as incremental updates, and rollback the state to any previous t on recovery, as we will see.

7.7 Achieving CL

7.7.1 Operator-local properties

The CL protocol is designed around the following local properties at each operator.

- an operator q can rely on its upstream operators to replay any input that resulted in state updates at q that were lost on failure, either because q could not finish computing the state update, or because the state update was lost before it could be persisted;

- an operator p must ensure it can replay any output to downstream operators that may have been lost on failure; to do so, p can rely on the first property, i.e. it can rely on its upstream operator to transitively provide lost information.

This properies must apply to all operators in the computation but inputs and outputs need special handling as they interact with components outside the system. As stated in our assumptions, inputs must be able to replay any subset of the input records requested by their immediately downstream operators. Consumers of the outputs are also able to request for output records to be replayed in case a failure at the consumer caused data loss; as we will see consumers are required to ackwnoledge timestamps for records that won't need to be replayed to allow garbage collection on state logs by CL.

7.7.2 Trivial fault-tolerance

Given the assumptions and properties discussed so far a trivial fault-tolerance mechanism can simply rely on the ability of the computation inputs to replay all input records from the start and achieve the consistency goal. While this approach would produce correct results, it would also cause all data processed before the failure to be performed again and can be prohibitively expensive and result in prolonged unavailability as the lost state is reconstructed from scratch.

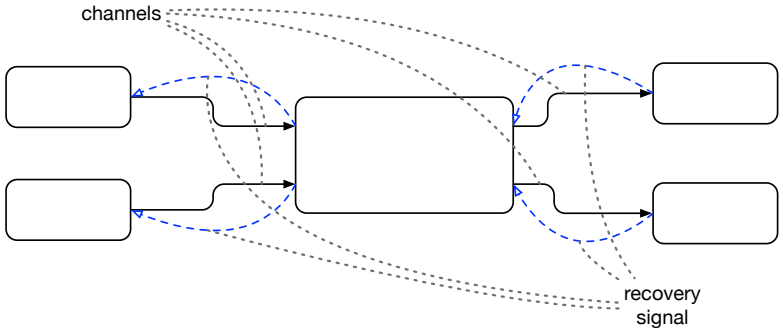


Figure 7.3: A fragment of a dataflow graph with recovery signals for CL.

7.7.3 Recording state updates

With the programming interface described in section 7.6 the system is able to asynchronously track and persist incremental state updates for each timestamp that is retired at an operator. When recovering from a failure, all operators can independently have access to a prefix of the state updates they recorded before the failure: some suffix of the state updates may have been lost because they are persisted asynchronously.

Each operator can determine which is the first timestamp t for which it has recovered incomplete state updates and it can request all upstream operators to replay their outputs (its inputs) starting from t. In CL this recovery signal flows on control edges as depicted in Figure 7.3: for each regular dataflow channel there is a recovery signal edge in the opposite direction.

This is an important improvement over the trivial fault-tolerance mechanism that replays the entire computation: except for the state updates that were generated in the moments before failure and were lost, most of the work performed before failure can be recovered without the need to re-process the entire inputs. Intuitively the recorded state updates act as a summary of the records that have been processed to produce them.

Because only a suffix of the state updates at each operator needs to be recomputed, it is possible for the computation to recover with little overhead and in a short amount of time (which depends on the amount

of work that was lost). The obvious downside of this approach is that CL would have to record the entire history of the state updates produced by each operator instance in the system: in a long-running dataflow computation over a large dataset this is likely a prohibitively large amount of data to maintain.

7.7.4 Reclaiming state updates

CL can successfully recover after a failure if all operators can replay all records that its immediate downstream operators require to maintain the properties described in subsection 7.7.1. Downstream operators can signal upstream a "recovery frontier": the earliest timestamp for which they still require upstream operators to be able to replay records in case of a failure. This frontier corresponds to the earliest timestamp associated with a state update that has not been acknowledged as successfully persisted by the state store (remember that the durable store performs state `put` requrests asynchronously).

In CL, this frontier information is communicated on the recovery signal channels of Figure 7.3. An operator can determine that it will not need detailed state update information for timestamps up to the lower bound of the recovery frontiers communicated by operators connected to its outputs. This recovery frontier lower bound advances as the computation progresses and downstream operators compute and persist more state updates.

State update history below the recovery frontier lower bound can be successfully compacted to reclaim memory and storage space. The compaction mechanism for traces described in Chapter 6 can be similarly applied here. If the state store persists new state updates soon after they are produced by their respective operators, the recovery frontier lower bound at every operator advances as the computation progresses; only detailed state update history for the most recent timestamps must be maintained, and all old state updates can be compacted and the corresponding data garbage collected. This ensures that the memory and state storage space used is proportional to the "working set size" of the operators and does not grow without bound.

7.7.5 CL

In CL, both recovery frontier updates and the recovery signal flow on the control edges introduced in Figure 7.3, operators compute and persist state updates, inform upstream operators of changes in the recovery frontier as they receive ackwonledgement from the state store, and compact and garbage collect their state updates based on the recovery frontier information they receive from downstream operators. Each operator instance can perform each of these tasks independently and concurrently with all other operator instances.

7.8 Implementation

We built CL as a library on top of the timely dataflow[MT] stream processing system using timestamp tokens to encode the recovery signals. Our library provides an interface to implement operators that comply with the operator model of Figure 7.2: the programmer provides the implementation of the state update (s) and output generation (o) functions.

A separate "persistence" thread for each timely dataflow worker receives state update `put` operations from local operator instances and asynchronously persists the state to the chosen state storage; operator instances perform state compaction based on the recovery frontier, as described in subsection 7.7.4, `put` compacted summaries of old state updates and inform the persistence thread of which state update history can be deleted.

7.9 Preliminary evaluation

As a preliminary evaluation of CL's performance impact on a realistic, albeit simple, data processing use case, we adapted the timely dataflow implementation of NEXMark query 5, that we open sourced as part of a related project [Hof+20]. A similar implementation (but not the same query) was used for the evaluation of timestamp tokens in subsection 4.5.4.

Our comparison baseline is an unmodified computation for NEXMark query 5 on timely dataflow, without any fault-tolerance features. This baseline is unable to recover after a failure without reprocessing the entire

input as no durability mechanism is present in timely dataflow. We implemented equivalent operator logic using the CL's operator programming interface (described in section 7.8).

For this experiment we simulated an input source able to replay inputs by restarting the NEXMark synthetic input generator from the timestamp corresponding to the recovery frontier of the first operator in the dataflow (the immediate downstream of the input). The output of the computation immediately acknowledges timestamps as they are completed: in a deployed system these acknowledgements may be delayed until the consumer of the outputs has been able to persist or otherwise process the output records. We ran preliminary experiment on an Intel Xeon E5-4650 v2 @2.40 GHz machine with 512 GiB of RAM.

Figure 7.1 shows p75 and maximum latency over time at a fixed 5000 records/sec offered load. Note that new records arrive continually at the inputs, even during the time CL is recovering a failure. There is no failure for the baseline, as timely dataflow does not have a fault-tolerance mechanism. For CL, we simulate worker failures at 200 and 400 seconds by terminating the processes. We then restart the computation and allow CL to recover and resume execution. The latency spikes at 200 and 400 seconds are CL performing recovery and catching up on new input that continued to arrive.

CL by design trades off some throughput (compared to the baseline) to maintain consistent steady state latency and to handle failures. To determine the throughput impact of CL we run a steady state experiment for NEXMark Q5 for both the baseline and CL implementation: we do not simulate any failure. We vary the offered load, and measure latency. Figure 7.4 shows p75, p99, and maximum latency for the baseline and CL over the entire duration of the experiments. In this experiment CL can sustain about half of the throughput of the baseline while maintaining interactive latencies (a latency over a few hunderd milliseconds indicates that the system is overloaded).

Preliminary strong scaling experiments display currently unexplained unexpected phoenomena possibly due to a communication bottleneck in the current implementation. More investigation is required to determine the throughput and latency trade-offs of CL in different configurations and for different data processing tasks.

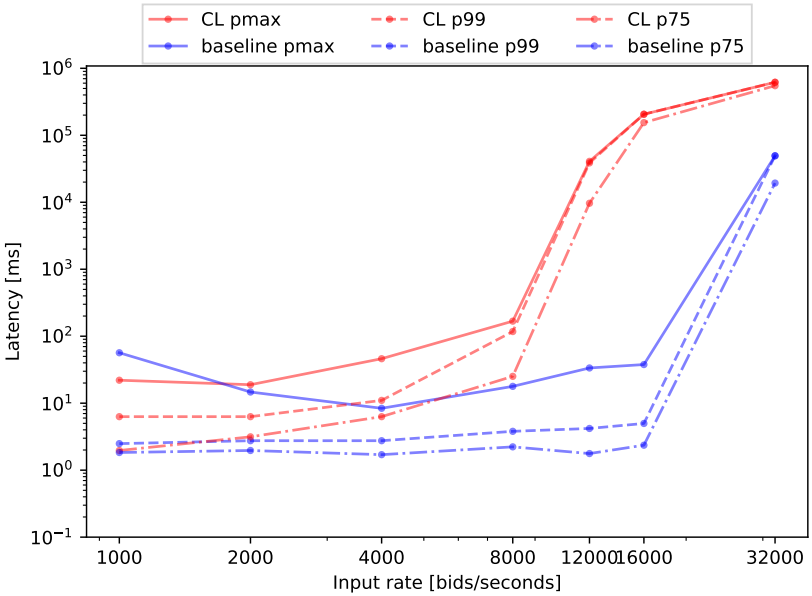


Figure 7.4: NexMark Q5 comparing latency of a non-fault-tolerant baseline with CL when the offered load varies in a configuration with three timely dataflow workers.

7.10 Re-scaling with CL

In Megaphone [Hof+18][1], our mechanism for live state-migration for distributed dataflow systems, we discuss how previous approaches to state migration based on state snapshots suffer from unavailability (or a latency spike) while the state snapshot is captured and redistributed across the workers in the systems.

This shortcoming of leveraging a fault tolerance mechanism for re-scaling and re-partitioning is alleviated by CL, which captures state continually as the computation progresses, leaving a small amount of state that needs to be captured synchronously when the re-scaling decision is made.

To enable re-scaling and re-partitioning we extended CL to capture the state for each operator instance in a number of separate bins, which act as state shards that can be moved across workers. When re-scaling without explicit re-partitioning (e.g. to handle a hot key), well-known consistent hasing-like strategies to assign bins to workers can be employed to minimize the number of bins that need to be moved between workers.

7.10.1 Coordinating compaction for rescaling

When CL is set up for rescaling, there is an additional constraint on how compaction is performed in addition to the considerations described in subsection 7.7.4. Across a re-partitioning operation, workers will become responsible for new state shards that were reassigned from other workers, which may have been performing compaction based on their local compaction frontier (based on the recovery frontier signals they received). On the other hand, a worker needs to be able to recover all shards at a unique frontier that it can communicate to its upstream operators to request replay of the inputs it lost.

In the absence of additional coordination no recovery frontier may be compatible with the state compaction that has been performed on local shards and shards re-assigned from other workers. This is because each worker processes the recovery signals independently, and will proceed independently with compaction. To ensure that any subset of state

[1] Megaphone is briefly discussed in section 5.3.

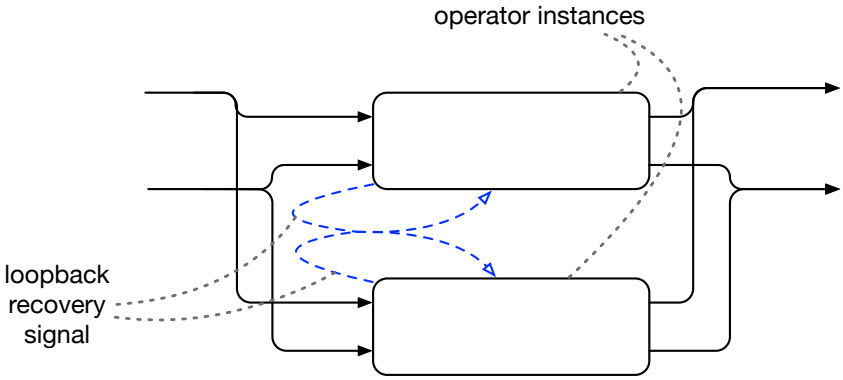


Figure 7.5: A dataflow operator with the additionoal recovery signal for rescaling with CL.

shards across the system can recover from a common recovery frontier, CL introduces an additional coordination signal to support rescaling.

Figure 7.5 depicts the additional coordination recovery signal for each operator; the figure shows two of possibly many instances of a single operator, the construction is repeated for each operator in the detaflow. The recovery signal from each logical operator is looped back to itself, ensuring that all operator instances can determine the lowest recovery signal across all other instances. This global recovery frontier is used to constrain the compaction of the update histories for the state shards, ensuring that they all can be subset to a common frontier when they are re-partitioned.

7.10.2 Rescaling while recovering

In certain deployment scenarios, it may make sense to recover in a different configuration than before the failure occurred. For example, if a node was permanently lost, it may be faster to resume execution using just the surviving nodes, potentially with degraded performance but with only a short downtime; likely shorter than waiting for a new node to be provisioned for the computation. To this end, CL supports redistributing state shards on recovery; the only requirement is that nodes have access

to storage for the state shards they inherited.

This allows advanced fault-tolerance configurations where the state backend for durability places one or more copies of the durable history on or near other nodes that participate in the computation, ensuring that n workers have quick access to the state history for the state shards of each other worker. This way recovering from n failures does not require any state transfer, as each worker in the new configuration can immediately inherit state shards for which it has local history copies.

7.11 Conclusions

This chapter presented a design for a fault-tolerance and dynamic re-scaling mechanism for distributed dataflow systems aimed at minimizing critical path synchronization. Timestamp tokens enabled the construction of the mechanism as a library on top of the timely dataflow[MT] stream processor, without changes to the underlying system.

By leveraging the data dependency information encoded in the per-tuple timestamps of time-aware systems, the durability process can be fully asynchronous, with each operator instance recording state updates independently. This results in a mechanism that does not require pausing the computation to collect consistent snapshots, thus avoiding latency spikes in steady state, as shown in section 7.9.

The mechanism also enables re-scaling, both as a planned operation or in response to a failure, where both are handled by the same durability and recovery protocol.

8 Verified progress tracking

This chapter is based on "Verified Progress Tracking for Timely Dataflow" [Bru+21] and references work done by Sára Decova in the context of her Master Thesis "Modelling and Verification of the Timely Dataflow Progress Tracking Protocol".

Formally specified and verified foundational software, such as operating systems and distributed runtimes, provide a solid a safe interface for building applications. The application developer can trust the programming interface and rely on it to build correct software, either just by having access to accurate and complete formal documentation or by building on the formal spec to verify the higher level software.

Streaming dataflow systems act as foundational components for many applications from data science to datacenter monitoring and management. Operators often rely on the low latency results to automate tasks: incorrect results can rapidly trigger incoherent automation that can cause outages, data loss, or worse.

In this chapter we present the formal specification and verification of the core coordination component of timely dataflow [MT], an high-throughput, low-latency, data-parallel time-aware dataflow system. As we have seen in Chapter 3, the coordination component ensures data is processed at the right time so that the outputs are correct. To achieve high expressiveness and performance, timely dataflow uses an intricate distributed protocol for tracking the computation's progress. We modeled this *progress tracking protocol* as a combination of two independent transition systems in the Isabelle/HOL proof assistant. We specified and verified the safety of the two components and of the combined protocol.

To this end, we identified abstract assumptions on dataflow programs that are sufficient for safety and were not previously formalized.

Timestamp tokens, described in Chapter 4, express a clean interface between operators and the system. This effort to formalize and verify the safety of the progress tracking protocol uses timestamp tokens as a basis for the formalization: using timestamp tokens we can precisely model what actions each instance of an operator can perform, in contrast with previous formalisation work [Aba+13] that pre-dates timestamp tokens.

8.1 Introduction

A *progress tracking mechanism* is a core component of the dataflow system. It receives information on outstanding timestamps from operator instances, exchanges this information with other system workers (cores, nodes) and disseminates up-to-date approximations of the frontiers to all operator instances.

The progress tracking mechanism must be correct. Incorrect approximations of the frontiers can result in subtle concurrency errors, which may only appear under certain load and deployment circumstances. In this work, we formally model in Isabelle/HOL and prove the safety of the progress tracking protocol of *Timely Dataflow* [Mur+13; Mur+16; MT], a time-aware dataflow programming model and a state-of-the-art streaming, data-parallel, distributed data processor.

In Timely Dataflow's progress tracking, worker-local and distributed coordination are intertwined, and the formal model must account for this asymmetry. Individual agents (operator instances) on a worker generate coordination updates that have to be asynchronously exchanged with all other workers, and then propagated locally on the dataflow structure to provide local coordination information to all other operator instances.

This is an additional (worker-local) dimension in the specification when compared to well-known distributed coordination protocols, such as Paxos [Lam02] and Raft [OO14], which focus on the interaction between symmetric communicating parties on different nodes. In contrast our environment model can be simpler, as progress tracking is designed to handle but not recover from fail-stop failures or unbounded pauses: upon crashes, unbounded stalls, or reset of a channel, the system stops without violating

safety. [1]

Abadi et al. [Aba+13] formalize and prove safety of the distributed exchange component of progress tracking in the TLA$^+$ Proof System. We present their *clocks protocol* through the lens of our Isabelle reformalization (Section 8.5) and show that it subtly fails to capture behaviors supported by Timely Dataflow [Mur+13; Mur+16]. We then significantly extend the formalized protocol (Section 8.6) to faithfully model Timely Dataflow's modern reference implementation [MT].

The above distributed protocol does not model the dataflow graph, operators, and timestamps within a worker. Thus, on its own it is insufficient to ensure that up-to-date frontiers are delivered to all operator instances. To this end, we formalize and prove the safety of the *local propagation* component (Section 8.7) of progress tracking, which computes and updates frontiers for all operator instances. Local propagation happens on a single worker, but operator instances act as independent asynchronous actors. For this reason, we also employ a state machine model for this component. Along the way, we identify sufficient criteria on dataflow graphs, that were previously not explicitly (or only partially) formulated for Timely Dataflow.

Finally, we combine the distributed component with local propagation (Section 8.8) and formalize the global safety property that connects initial timestamps to their effect on the operator frontier. Specifically, we prove that our combined protocol ensures that frontiers always constitute safe lower bounds on what timestamps may still appear on the operator inputs.

8.2 Related Work

Data management systems verification Timely Dataflow is a system that supports low-latency, high-throughput data-processing applications. Higher level libraries (such as [McS+13] and the one described in Chapter 6) and SQL abstractions [Mat] built on Timely Dataflow support high performance incremental view maintenance for complex queries over large datasets. Verification and formal methods efforts in the data management and processing space have focused on SQL and query-language

[1] Systems based on Timely Dataflow and progress tracking can recover by re-starting the protocol.

semantics [BC19; DOT20; Chu+17] and on query runtimes in database management systems [Mal+10; Ben+18].

Distributed systems verification Timely Dataflow is a distributed, concurrent system: our modeling and proof techniques are based on the widely accepted state machine model and refinement approach as used, e.g., in the TLA$^+$ Proof System [Cha+10] and Ironfleet [Haw+15]. Recent work focuses on proving consistency and safety properties of distributed storage systems [LBC16; Gom+17; Han+20] and providing tools for the implementation and verification of general distributed protocols [Jun+18; Spr+20] leveraging domain-specific languages [Wil+15; SWT18] and advanced type systems [HBK20].

Model of Timely Dataflow Abadi and Isard [AI15a] define abstractly the semantics of a Timely Dataflow programming model [Mur+13]. Our work is complementary; we concretely compute their *could-result-in* relation (Section 8.8) and formally model the implementation's core component.

8.3 Timely Dataflow and Progress Tracking

Our formal model follows the progress tracking protocol of the modern Rust implementation of Timely Dataflow [MT] using timestamp tokens, as described in Chapter 4. The protocol has evolved from the one reported as part of the classic implementation Naiad [Mur+13]. Here, we provide an informal overview of the basic notions, for the purpose of supporting the presentation of our formal model and proofs.

Dataflow graph Timely Dataflow is a time-aware dataflow system, as described in Chapter 3. In Timely Dataflow each worker in the system runs an instance of the entire dataflow graph, so all operators have the same number of instances. Workers run independently and only communicate through reliable message queues—they act as communicating sequential processes [Hoa78]. Each worker alternately executes the progress tracking protocol and the operator's processing logic. Figure 8.1 shows a Timely Dataflow operator and the related concepts described in this section.

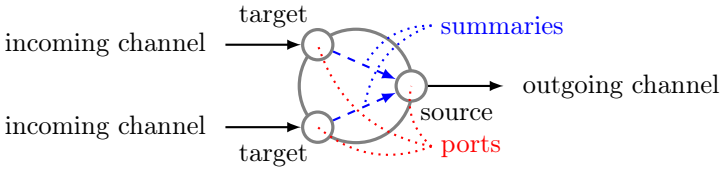


Figure 8.1: A Timely Dataflow operator.

Pointstamps A pointstamp represents a datum at rest at an operator, or in motion on one of the channels. A pointstamp (l, t) refers to a location l in the dataflow and a timestamp t. As we have seen in Chapter 2 and Chapter 3, timestamps encode a semantic (causal) grouping of the data. Timestamps are usually tuples of positive integers, but can be of any type for which a partial order $\preceq$ is defined.

Locations and summaries Each operator has an arbitrary number of input and output ports, which are locations. An operator instance receives new data through its input ports, or target locations, performs processing, and produces data through its output ports, or source locations. A dataflow channel is an edge from a source to a target. Internal operator connections are edges from a target to a source, which are additionally described by one or more summaries: the minimal increment to timestamps applied to data processed by the operator.[2]

Frontiers Frontiers are a lower bound on the timestamps that may appear at the operator instance inputs, as we have seen in subsection 3.1.6. The progress tracking protocol tracks the system's pointstamps and summarizes them to one frontier per operator port. In Timely Dataflow a frontier is represented by an antichain F indicating that the operator may still receive any timestamp t for which $\exists t' \in F.\ t' \preceq t$.

Progress tracking Progress tracking computes frontiers in two steps. A distributed component *exchanges* pointstamp changes (Sections 8.5 and 8.6) to construct an approximate, conservative view of all the pointstamps present in the system. Workers use this global view to locally *propagate*

[2]Internal summaries are the internal dependencies of the time-aware dataflow model described in subsection 3.1.5.

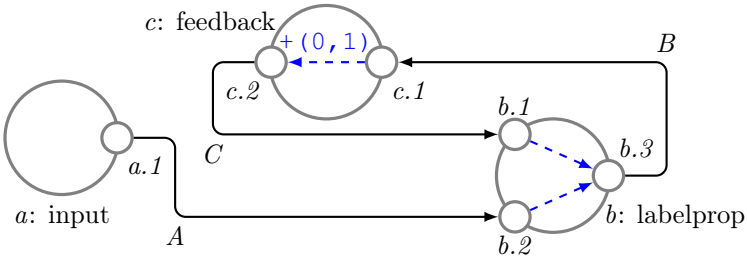


Figure 8.2: A timely dataflow that computes weakly connected components.

changes on the dataflow graph (Section 8.7) and update the frontiers at the operator input ports. The combined protocol (Section 8.8) asynchronously executes these two components.

8.4 Running Example: Weakly Connected Components by Propagating Labels

Figure 8.2 shows a dataflow that computes weakly connected components (WCC) by assigning integer labels to vertices in a graph, and propagating the lowest label seen so far by each vertex to all its neighbors. The input graph is initially sent by operator *a* as a stream of edges `(s,d)` with timestamp `(0,0)`. Each input port has an associated sharding function to determine which data should be sent to which operator instance: port *b.2* shards the incoming edges `(s,d)` by `s`.

The input operator *a* will continue sending additional edges in the graph as they appear, using increasing timestamps by incrementing one coordinate: `(1,0)`, `(2,0)`, etc. The computation is tasked with reacting to these changes and performing incremental re-computation to produce correct output for each of these input graph versions. The first timestamp coordinate represents logical consistency boundaries for the input and output of the program. We will use the second timestamp coordinate to track the progress of the unbounded iterative algorithm.

The operator *a* starts with a pointstamp (*a.1*, `(0,0)`) on port *a.1*, representing its intent to send data with that timestamp through the connected channel. When it sends messages on channel *A*, these are rep-

resented by pointstamps on the port *b.2*; e.g., (*b.2*, `(0,0)`) for the initial timestamp `(0,0)`. When it ceases sending data for a certain timestamp, e.g., `(0,0)`, operator *a* drops the corresponding pointstamp on port *a.1*. The frontier at *b.2* reflects whether pointstamps with a certain timestamp are present at either *a.1* or *b.2*: when they both become absent (when all messages are delivered) each instance of *b* notices that its frontier has advanced and determines it has received its entire share of the input (the graph) for a timestamp.

Each instance of *b* starts with a pointstamp on *b.3* at timestamp `(0,0)`; when it has received its entire share of the input, for each vertex with label `x` and each of its neighbors `n`, it sends `(n,x)` at timestamp `(0,0)`. This stream then traverses operator *c*, that increases the timestamp associated to each message by `(0,1)`, and reaches port *b.1*, which shards the incoming tuples `(n,x)` by `n`. Operator *b* inspects the frontier on *b.1* to determine when it has received all messages with timestamp `(0,1)`. These messages left *b.3* with timestamp `(0,0)`. The progress tracking mechanism will correctly report the frontier at *b.1* by taking into consideration the summary between *c.1* and *c.2*.

Operator *b* collects all label updates from *b.1* and, for those vertices that received a value that is smaller than the current label, it updates internal state and sends a new update via *b.3* with timestamp `(0,1)`. This process then repeats with increasing timestamps, `(0,2)`, `(0,3)`, etc., for each trip around the loop, until ultimately no new update message is generated on port *b.3* by any of the operator instances, for a certain family of timestamps (t_1, t_2) with a fixed t_1 corresponding to the input version being considered. Operator *b* determines it has correctly labeled all connected components for a given t_1 when the frontier at *b.1* does not contain a (t_1, t_2) such that $t_2 \preceq$ the graph's diameter. In practice, once operator *b* determines it has computed the output for a given t_1, the operator would also send the output on an additional outgoing channel to deliver it to the user. Later, operator *b* continues processing for further input versions, indicated by increasing t_1, with timestamps $(t_1, 0)$, $(t_1, 1)$, etc.

8.5 The Clocks Protocol

In this section, we present Abadi et al.'s approach to modeling the distributed component of progress tracking [Aba+13], termed the *clocks protocol*. Instead of showing their TLA$^+$ Proof System formalization, we present our re-formalization of the protocol in Isabelle. Thereby, this section serves as an introduction to both the protocol and the relevant Isabelle constructs.

The clocks protocol is a distributed algorithm to track existing pointstamps in a dataflow. It models a finite set of workers. Each worker stores a (finite) multiset of pointstamps as seen from its perspective and shares updates to this information with all other workers. The protocol considers workers as black boxes, i.e., it does *not* model their dataflow graph, locations, and timestamps. We extend the protocol to take these components into account in Section 8.7.

In Isabelle, we use the type variable $'w :: \mathit{finite}$ to represent workers. We assume that $'w$ belongs to the *finite* type class, which assures that $'w$'s universe is finite. Similarly, we model pointstamps abstractly by $'p :: \mathit{order}$. The *order* type class assumes the existence of a partial order $\leqslant :: 'p \Rightarrow 'p \Rightarrow \mathit{bool}$ (and the corresponding strict order $<$).

We model the protocol as a transition system that acts on configurations given as follows:

record $('w :: \mathit{finite}, 'p :: \mathit{order})\ \mathit{conf} =$
  rec :: $'p\ \mathit{zmset}$
  msg :: $'w \Rightarrow 'w \Rightarrow 'p\ \mathit{zmset}\ \mathit{list}$
  temp :: $'w \Rightarrow 'p\ \mathit{zmset}$
  glob :: $'w \Rightarrow 'p\ \mathit{zmset}$

Here, rec c denotes the global multiset of pointstamps (or records) that are present in a system's configuration c. We use the type $'p\ \mathit{zmset}$ of *signed multisets* [BFT17]. An element $M :: 'p\ \mathit{zmset}$ can be thought of as a function of type $'p \Rightarrow \mathit{int}$, which is non-zero only for finitely many values. (In contrast, an unsigned multiset $M :: 'p\ \mathit{mset}$ corresponds to a function of type $'p \Rightarrow \mathit{nat}$.) Signed multisets enjoy nice algebraic properties; in particular, they form a group. This significantly simplifies the reasoning about subtraction. However, rec c will always store only non-negative pointstamp counts. The other components of a configuration c are

- the progress message queues $\mathsf{msg}\ c\ w\ w'$, which denote the progress update messages sent from worker w to worker w' (not to be confused with data messages, which are accounted for in $\mathsf{rec}\ c$ but do not participate in the protocol otherwise);
- the temporary changes $\mathsf{temp}\ c\ w$ in which worker w stores changes to pointstamps that it might need to communicate to other workers; and
- the local approximation $\mathsf{glob}\ c\ w$ of $\mathsf{rec}\ c$ from the perspective of worker w (we use Abadi et al. [Aba+13]'s slightly misleading term glob for the worker's *local* view on the global state).

In contrast to rec, these components may contain a negative count $-i$ for a pointstamp p, which denotes that i occurrences of p have been discarded.

The following predicate characterizes the protocol's initial configurations. We write $\{\#\}_z$ for the empty signed multiset and $M\ \#_z\ p$ for the count of pointstamp p in a signed multiset M.

definition Init :: $('w, 'p)\ conf \Rightarrow bool$ **where**
$\mathsf{Init}\ c = (\forall p.\ \mathsf{rec}\ c\ \#_z\ p \geqslant 0) \wedge (\forall w\ w'.\ \mathsf{msg}\ c\ w\ w' = [\,]) \wedge$
$(\forall w.\ \mathsf{temp}\ c\ w = \{\#\}_z) \wedge (\forall w.\ \mathsf{glob}\ c\ w = \mathsf{rec}\ c)$

In words: all global pointstamp counts in rec must be non-negative and equal to each worker's local view glob; all message queues and temporary changes must be empty.

Referencing our WCC example described in section 8.4, the clocks protocol is the component in charge of distributing pointstamp changes to other workers. When one instance of the input operator a ceases sending data for a certain family of timestamps `(`t_1`,0)` it drops the corresponding pointstamp: the clocks protocol is in charge of *exchanging* this information with other workers, so that they can determine when all instances of a have ceased producing messages for a certain timestamp. This happens for all pointstamp changes in the system, including pointstamps that represent messages in-flight on channels.

The configurations evolve via one of three actions:

perf_op: A worker may perform an operation that causes a change in pointstamps. Changes may remove certain pointstamps and add others. They are recorded in rec and temp.

send_upd: A worker may broadcast some of its changes stored in temp to all other workers.

recv_upd: A worker may receive an earlier broadcast and update its local view glob.

Overall, the clocks protocol aims to establish that glob is a safe approximation for rec. Safe means here that no pointstamp in rec is less than any of glob's minimal pointstamps. To achieve this property, the protocol imposes a restriction on which new pointstamps may be introduced in rec and which progress updates may be broadcast. This restriction is the *uprightness* property that ensures that a pointstamp can only be introduced if simultaneously a smaller (supporting) pointstamp is removed. Formally, a signed multiset of pointstamps is upright if every positive entry is accompanied by a smaller negative entry:

definition supp :: $'p$ *zmset* $\Rightarrow$ $'p \Rightarrow$ *bool* **where** supp M p = $(\exists p' < p.\ M\ \#_z\ p' < 0)$
definition upright :: $'p$ *zmset* $\Rightarrow$ *bool* **where** upright M = $(\forall p.\ M\ \#_z\ p > 0 \longrightarrow$ supp M $p)$

Abadi et al. [Aba+13] additionally require that the pointstamp p' in supp's definition satisfies $\forall p'' \leqslant p'.\ M\ \#_z\ p'' \leqslant 0$. The two variants of upright are equivalent in our formalization because signed multisets are finite and thus minimal elements exist even without $\leqslant$ being well-founded. The extra assumption on p' is occasionally useful in proofs.

In practice, uprightness means that operators are only allowed to transition to pointstamps forward in time, and cannot re-introduce pointstamps that they relinquished. This is necessary to ensure that the frontiers always move to later timestamps and remain a conservative approximation of the pointstamps still present in the system. An advancing frontier triggers computation in some of the dataflow operators, for example to output the result of a time-based aggregation: this should only happen once all the relevant incoming data has been processed. This is the intuition behind the safety property of the protocol, Safe, discussed later in this section.

Figure 8.3 defines the three protocol actions formally as transition relations between an old configuration c and a new configuration c' along with the definition of the overall transition relation Next, which in addition to

definition perf_op :: $'w \Rightarrow$ $'p$ *mset* $\Rightarrow$ $'p$ *mset* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ *bool* **where**
perf_op w Δ_{neg} Δ_{pos} c c' = let $\Delta = \Delta_{pos} - \Delta_{neg}$ in ($\forall p$. Δ_{neg} # $p \leqslant$ rec c #$_z$ p) $\wedge$ upright Δ $\wedge$
$c' = c(|$rec = rec $c + \Delta$, temp = (temp c)(w := temp c $w + \Delta$)$|)$

definition send_upd :: $'w \Rightarrow$ $'p$ *set* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ *bool* **where**
send_upd w P c c' = let $\gamma = \{\#p \in\#_z$ temp c w. $p \in P\#\}$ in
$\gamma \neq \{\#\}_z \wedge$ upright (temp c $w - \gamma$) $\wedge$
$c' = c(|$msg = (msg c)(w := $\lambda w'$. msg c w $w' \cdot [\gamma]$), temp = (temp c)(w := temp c $w - \gamma$)$|)$

definition recv_upd :: $'w \Rightarrow$ $'w \Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ *bool* **where**
recv_upd w w' c c' = msg c w $w' \neq$ [] $\wedge$
$c' = c(|$msg = (msg c)(w := (msg c w)(w' := tl (msg c w w'))),
glob = (glob c)(w' := glob c w' + hd (msg c w w'))$|)$

definition Next :: ($'w$, $'p$) *conf* $\Rightarrow$ ($'w$, $'p$) *conf* $\Rightarrow$ *bool* **where**
Next c c' = ($c = c'$) $\vee$ ($\exists w$ Δ_{neg} Δ_{pos}. perf_op w Δ_{neg} Δ_{pos} c c') $\vee$
($\exists w$ P. send_upd w P c c') $\vee$ ($\exists w$ w'. recv_upd w w' c c')

Figure 8.3: Transition relation of Abadi et al.'s clocks protocol

performing one of the actions may stutter, i.e., leave $c' = c$ unchanged. The three actions take further parameters as arguments, which we explain next.

The action perf_op is parameterized by a worker w and two (unsigned) multisets Δ_{neg} and Δ_{pos}, corresponding to negative and positive pointstamp changes. The action's overall effect on the pointstamps is thus $\Delta = \Delta_{pos} - \Delta_{neg}$. Here and elsewhere, subtraction expects signed multisets as arguments and we omit the type conversions from unsigned to signed multisets (which are included in our Isabelle formalization). The action is only enabled if its parameters satisfy two requirements. First, only pointstamps present in rec may be dropped, and thus the counts from Δ_{neg} must be bounded by the ones from rec. (Arguably, accessing rec is problematic for distributed workers. We rectify this modeling deficiency in Section 8.6.) Second, Δ must be upright, which ensures that we will never introduce a pointstamp that is lower than any pointstamp in rec. If these requirements are met, the action can be performed and

will update both rec and temp with Δ (expressed using Isabelle's record and function update syntax).

The action send_upd is parameterized by a worker (sender w) and a set of pointstamps P, the outstanding changes to which, called γ, we want to broadcast. The key requirement is that the still unsent changes remain upright. Note that it is always possible to send all changes or all positive changes in temp, because any multiset without a positive change is upright. The operation enqueues γ in all message queues that have w as the sender. We model first-in-first-out queues as lists, where enqueuing means appending at the end ($\_ \cdot [\_]$).

Finally, the action recv_upd is parameterized by two workers (sender w and receiver w'). Given a non-empty queue msg $c\ w\ w'$, the action dequeues the first message (head hd gives the message, tail tl the queue's remainder) and adds it to the receiver's glob.

An execution of the clocks protocol is an infinite sequence of configurations. Infinite sequences of elements of type *'a* are expressed in Isabelle using the coinductive datatype (short codatatype) of streams defined as **codatatype** *'a stream* = Stream *'a* (*'a stream*). We can inspect a stream's head and tail using the functions shd :: *'a stream* ⇒ *'a* and stl :: *'a stream* ⇒ *'a stream*. Valid protocol executions satisfy the predicate Spec, i.e., they start in an initial configuration and all neighboring configurations are related by Next:

definition Spec :: (*'w*, *'p*) *conf stream* ⇒ *bool* **where**
  Spec s = now Init s ∧ alw (relates Next) s

The operators now and relates lift unary and binary predicates over configurations to executions by evaluating them on the first one or two configurations respectively: now $P\ s = P$ (shd s) and relates $R\ s = R$ (shd s) (shd (stl s)). The coinductive operator alw resembles a temporal logic operator: alw $P\ s$ holds if P holds for all suffixes of s.

coinductive alw :: (*'a stream* ⇒ *bool*) ⇒ *'a stream* ⇒ *bool* **where**
  $P\ s \longrightarrow$ alw P (stl s) $\longrightarrow$ alw $P\ s$

We use the operators now, relates, and alw not only to specify valid execution, but also to state the main safety property. Moreover, we use the predicate vacant to express that a pointstamp (and all smaller pointstamps) are not present in a signed multiset:

definition vacant :: *'p zmset* ⇒ *'p* ⇒ *bool* **where** vacant M p = $(\forall p' \leqslant p.\ M\ \#_z\ p' = 0)$

Safety states that if any worker's glob becomes vacant up to some pointstamp, then that pointstamp and any lesser ones do not exist in the system, i.e., are not present in rec (and will remain so). Thus, safety allows workers to learn locally, via glob, something about the system's global state rec, namely that they will never encounter certain pointstamps again. Formally:

definition Safe :: (*'w*, *'p*) *conf stream* ⇒ *bool* **where**
Safe s = $(\forall w\ p.$ now $(\lambda c.$ vacant (glob $c\ w$) $p)\ s \longrightarrow$
alw (now $(\lambda c.$ vacant (rec c) $p)\ s$))

lemma *safe:* Spec $s \longrightarrow$ alw Safe s

Proof 3 (Proof Sketch) *We prove safety following Abadi et al. [Aba+13]. First, we establish three invariants by showing that* Next *preserves them:*

1. rec *only contains positive entries*
2. rec *is the sum of any worker w's* glob *and its* incoming information info $c\ w = \sum_{w'}$(temp $c\ w' + \sum_{M \in \text{set (msg } c\ w'\ w)} M$)*, that is the sum of all workers'* temp *and all* msg *directed towards w*
3. *any worker w's incoming information is upright*

We then show that whenever rec *becomes vacant up to some pointstamp p, then it forever stays vacant up to p. Thus, we can eliminate the "inner" occurrence of* alw *from the definition of* Safe*. The remaining property follows by contradiction, i.e., by assuming a non-zero count for some pointstamp p in* rec*, up to which some worker w's* glob *is vacant. Invariants 1 and 2 imply that w's incoming information has a positive count for p. Because it is upright by invariant 3, w's incoming information must also contain a smaller pointstamp $q < p$ with a negative count. But w's* glob *count for q must be zero (recall that w's* glob *is vacant up to p), which together with invariant 2 implies that* rec *has a negative count at q. This contradicts invariant 1.*

Having established safety, we also prove a second important property of glob formalized by Abadi et al.: monotonicity. This property states that once glob becomes vacant upto some pointstamp p, it will forever stay so:

definition Mono :: $('w, 'p)$ *conf* $\Rightarrow$ $('w, 'p)$ *conf* $\Rightarrow$ *bool* **where**
Mono c c' = $(\forall w\ p.$ vacant (glob c w) $p \longrightarrow$ vacant (glob c' w) $p)$

lemma *mono:* Spec $s \longrightarrow$ alw (relates Mono) s

Establishing glob's monotonicity is significantly more difficult than proving the same property for rec, which we have used in the proof of safety. New positive entries in rec can only be introduced in the perf_op transition, where they are guarded by smaller negative changes due to the uprightness requirement. In contrast, glob is altered in the recv_upd transition, where it is far less clear a priori why this step cannot introduce pointstamps up to which glob is vacant. The key idea, again due to Abadi et al., to establish glob's monotonicity is to generalize the notion of uprightness and show that all individual messages from msg satisfy the generalized notion. Abadi et al. call the generalized notion *beta uprightness*. It allows positive pointstamp entries from a message $M :: 'p\ zmset$ to be supported not only by smaller negative pointstamp entries in M itself, but also by negative entries in another multiset $N :: 'p\ zmset$.

definition beta_upright :: $'p\ zmset \Rightarrow 'p\ zmset \Rightarrow$ *bool* **where**
beta_upright M N = $(\forall p.\ M \#_z p > 0 \longrightarrow (\exists p' < p.\ M \#_z p' < 0 \vee N \#_z p' < 0))$

We do not describe in detail how beta uprightness helps with monotonicity, but the main step is to establish the invariant that all messages M from msg are beta upright with respect to N being the sum of messages following M in msg and the sender's temp.

Overall, we have replicated the formalization of Abadi et al.'s clocks protocol and proofs of its safety and the monotonicity of glob, each worker's approximated view of the system's pointstamps. Their protocol accurately models the implementation of the progress tracking protocol's distributed component in Timely Dataflow's original implementation Naiad with one subtle exception. The Naiad API (`OnNotify`, `SendBy`) allows an operator to repeatedly send data messages through its output port, which generates pointstamps at the receiver, without requiring that a pointstamp on the output port is decremented. This can result in a perf_op transition that is not upright.[3] Additionally, the modern reference implementation of Timely Dataflow in Rust is more expressive

[3] We refer here to locations as presented in Section 8.3. The model in Naiad is slightly different: there is no notion of ports, and pointstamp locations are either operators

than Naiad, and permits multiple operations that result in non-upright changes. We address and correct this limitation of the clocks protocol in Section 8.6.

One example of an operator that expresses behavior that results in non-upright changes is the input operator a in the WCC example. This operator may be reading data from an external source, and as soon as it receives new edges, it can forward them with the current pointstamp $(a.1, (t_1, 0))$. This operator may be invoked multiple times, and perform this action repeatedly, until it determines from the external source that it should mark a certain timestamp as complete by dropping the pointstamp. All of these intermediate actions that send data at $(t_1, 0)$ are not upright, as sending messages creates new pointstamps on the message targets, without dropping a smaller pointstamp that can support the postive change.

8.6 Exchanging Progress

As outlined in the previous section, the clocks protocol is not flexible enough to capture executions with non-upright changes, which are desired and supported by concrete implementations of Timely Dataflow. At the same time, the protocol captures behaviors that are not reasonable in practice. Specifically, the clocks protocol does not separate the worker-local state from the system's global state. The perf_op transition, which is meant to be executed by a single worker, uses the global state to check whether the transition is enabled and simultaneously updates the global state rec as part of the transition. In particular, a single perf_op transition allows a worker to drop a pointstamp that in the real system "belongs" to a different worker w and simultaneously consistently updates w's state. In concrete implementations of Timely Dataflow, workers execute perf_op's asynchronously, and thus can only base the transition on information that is locally available to them.

Our modified model of the protocol, called *exchange*, resolves both issues. As the first step, we split the rec field into worker-local signed multisets caps of pointstamps, which we call *capabilities* as they indicate the

or edges. A straightforward translation of the Naiad model interprets pointstamps on operators as pointstamps on their source port, and pointstamps on edges become pointstamps on the associated target ports.

possibility for the respective worker to emit these pointstamps. Workers may transfer capabilities to other workers. To do so, they asynchronously send capabilities as data messages to a central multiset data of pairs of workers (receivers) and pointstamps. We arrive at the following updated type of configurations:

record $('w :: \mathit{finite}, 'p :: \mathit{order})\ \mathit{conf} =$
  caps :: $'w \Rightarrow 'p\ \mathit{zmset}$
  data :: $('w \times 'p)\ \mathit{mset}$
  msg :: $'w \Rightarrow 'w \Rightarrow 'p\ \mathit{zmset}\ \mathit{list}$
  temp :: $'w \Rightarrow 'p\ \mathit{zmset}$
  glob :: $'w \Rightarrow 'p\ \mathit{zmset}$

Including this fine-grained view on pointstamps will allow workers to make transitions based on worker-local information. The entirety of the system's pointstamps, rec, which was previously part of the configuration and which the protocol aims to track, can be computed as the sum of all the workers' capabilities and data's in-flight pointstamps.

definition rec :: $('w, 'p)\ \mathit{conf} \Rightarrow 'p\ \mathit{zmset}$ **where** rec $c =$
$(\sum_w \mathsf{caps}\ c\ w) + \mathsf{snd}\ `\#\ \mathsf{data}\ c$

Here, the infix operator $`\#$ denotes the image of a function over a multiset with resulting counts given by $(f\ `\#\ M)\ \#\ x = \sum_{y \in \{y \in \# M \mid f\ y = x\}} M\ \#\ y$.

The exchange protocol's initial state allows workers to start with some positive capabilities. Each worker's glob must correctly reflect all initially present capabilities.

definition Init :: $('w, 'p)\ \mathit{conf} \Rightarrow \mathit{bool}$ **where**
  Init $c = (\forall w\ p.\ \mathsf{caps}\ c\ w\ \#_z\ p \geqslant 0) \wedge \mathsf{data}\ c = \{\#\} \wedge$
    $(\forall w\ w'.\ \mathsf{msg}\ c\ w\ w' = [\,]) \wedge (\forall w.\ \mathsf{temp}\ c\ w = \{\#\}_z) \wedge$
$(\forall w.\ \mathsf{glob}\ c\ w = \mathsf{rec}\ c)$

The transition relation of the exchange protocol, shown in Figure 8.4, is similar to that of the clocks protocol. We focus on the differences between the two protocols. First, the exchange protocol has an additional transition recv_cap to receive a previously sent capability. The transition removes a pointstamp from data and adds it to the receiving worker's capabilities.

The perf_op transition resembles its homonymous counterpart from the clocks protocol. Yet, the information flow is more fine grained. In

definition recv_cap :: $'w \Rightarrow 'p \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow$ $bool$ **where**
  recv_cap $w\ p\ c\ c' = (w, p) \in\#$ data $c \wedge$
    $c' = c(\!|$caps $=$ (caps c)$(w :=$ caps $c\ w + \{\#p\#\}_z)$, data $=$ data $c -$ $\{\#(w, p)\#\}|\!)$

definition perf_op :: $'w \Rightarrow 'p\ mset \Rightarrow ('w \times 'p)\ mset \Rightarrow 'p\ mset \Rightarrow$
    $('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  perf_op $w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c' =$
    $(\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z) \wedge (\forall p.\ \Delta_{neg}\ \#\ p \leqslant$ caps $c\ w\ \#_z\ p) \wedge$
    $(\forall (w', p) \in\#\ \Delta_{data}.\ \exists p' < p.$ caps $c\ w\ \#_z\ p' > 0) \wedge$
    $(\forall p \in\#\ \Delta_{self}.\ \exists p' \leqslant p.$ caps $c\ w\ \#_z\ p' > 0) \wedge$
    $c' = c(\!|$caps $=$ (caps c)$(w :=$ caps $c\ w + \Delta_{self} - \Delta_{neg})$, data $=$ data $c + \Delta_{data}$,
      temp $=$ (temp c)$(w :=$ temp $c\ w +$ (snd '$\#\ \Delta_{data} + \Delta_{self} - \Delta_{neg}$))$|\!)$

definition send_upd :: $'w \Rightarrow 'p\ set \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow$ $bool$ **where**
  send_upd $w\ P\ c\ c' =$ let $\gamma = \{\#p \in\#_z$ temp $c\ w.\ p \in P\#\}$ in
    $\gamma \neq \{\#\}_z \wedge$ justified (caps $c\ w$) (temp $c\ w - \gamma$) $\wedge$
    $c' = c(\!|$msg $=$ (msg c)$(w := \lambda w'.$ msg $c\ w\ w' \cdot [\gamma])$, temp $=$ (temp c)$(w :=$ temp $c\ w - \gamma)|\!)$

definition recv_upd :: $'w \Rightarrow 'w \Rightarrow ('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow$ $bool$ **where**
  recv_upd $w\ w'\ c\ c' =$ msg $c\ w\ w' \neq [] \wedge$
    $c' = c(\!|$msg $=$ (msg c)$(w :=$ (msg $c\ w$)$(w' :=$ tl (msg $c\ w\ w'$))),
      glob $=$ (glob c)$(w' :=$ glob $c\ w' +$ hd (msg $c\ w\ w'$))$|\!)$

definition Next :: $('w, 'p)\ conf \Rightarrow ('w, 'p)\ conf \Rightarrow bool$ **where**
  Next $c\ c' = (c = c') \vee (\exists w\ p.$ recv_cap $w\ p\ c\ c') \vee$
    $(\exists w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}.$ perf_op $w\ \Delta_{neg}\ \Delta_{data}\ \Delta_{self}\ c\ c') \vee$
    $(\exists w\ P.$ send_upd $w\ P\ c\ c') \vee (\exists w\ w'.$ recv_upd $w\ w'\ c\ c')$

Figure 8.4: Transition relation of the exchange protocol

particular, the transition is parameterized by a worker w and three multisets of pointstamps. As in the clocks protocol, the multiset Δ_{neg} represents negative changes to pointstamps. Only pointstamps for which w owns a capability in caps may be dropped in this way. The other two multisets Δ_{data} and Δ_{self} represent positive changes. The multiset Δ_{data} represents positive changes to other workers' capabilities—the receiving worker is stored in Δ_{data}. These changes are not immediately applied to the other worker's caps, but are sent via the data field. The multiset Δ_{self} represents positive changes to w's capabilities, which are applied immediately applied to w's caps. The separation between Δ_{data} and Δ_{self} is motivated by different requirements on these positive changes to pointstamps that we prove to be sufficient for safety. To send a positive capability to another worker, w is required to hold a positive capability for a strictly smaller pointstamp. In contrast, w can create a new capability for itself, if it is already holding a capability for the very same (or a smaller) pointstamp. In other words, w can arbitrarily increase the multiset counts of its own capabilities. Note that, unlike in the clocks protocol, there is no requirement of uprightness and, in fact, workers are not required to perform negative changes at all. Of course, it is useful for workers to perform negative changes every now and then so that the overall system can make progress.

The first condition in perf_op, namely $\Delta_{data} \neq \{\#\} \vee \Delta_{self} - \Delta_{neg} \neq \{\#\}_z$, ensures that the transition changes the configuration. In the exchange protocol, we also include explicit stutter steps in the Next relation ($c = c'$) but avoid them in the individual transitions.

Sending (send_upd) and receiving (recv_upd) progress updates works precisely as in the clocks protocol except for the condition on what remains in the sender's temp highlighted in gray in Figure 8.4. Because we allowed perf_op to perform non-upright changes, we can no longer expect the contents of temp to be upright. Instead, we use the predicate justified, which offers three possible justifications for positive entries in the signed multiset M (in contrast to upright's sole justification of being supported in M):

definition justified :: *'p zmset* ⇒ *'p zmset* ⇒ *bool* **where**
  justified C M = ($\forall p.\ M \#_z p > 0 \longrightarrow$ supp M $p \vee (\exists p' < p.\ C \#_z p' > 0) \vee M \#_z p < C \#_z p$)

Thus, a positive count for pointstamp p in M may be either

- supported in M, i.e., in particular every upright change is justified, or
- justified by a smaller pointstamp in C, which we think of as the sender's capabilities, or
- justified by p in C, with the requirement that p's count in M is smaller than p's count in C.

The definitions of valid executions Spec and the safety predicate Safe are unchanged compared to the clocks protocol. Also, we prove precisely the same safety property *safe* following a similar proof structure. The main difference is that uprightness invariant 3 is replaced by the statement that every worker's incoming information is justified with respect to pointstamps present in rec, i.e. $\forall w.$ justified (rec c) (info c w). It is more tedious to reason about pointstamps being justified compared with being upright due to the three-way case distinction that is usually necessary. These case distinctions occur when establishing the above invariant, but also in the contradiction proof establishing safety. The contradiction proof proceeds as before by assuming a non-zero count for some pointstamp p in rec, up to which some worker w's glob is vacant. Crucially, p is now additionally and without loss of generality assumed to be a minimal pointstamp with this property. By invariants 1 and 2, we deduce that w's incoming information has a positive count for p. Because it is justified by the new invariant 3, we perform the case distinction on the justification. If p is supported in w's incoming information, we proceed as in the clocks protocol. If p is justified by a positive count for a strictly smaller pointstamp in rec, we obtain a contradiction to p's minimality. Finally, if p's multiplicity in w's incoming information is strictly smaller than p's multiplicity in rec, invariant 2 tells us that p must have a positive count in glob, which contradicts the assumption of glob being vacant up to p.

We prove glob's monotonicity for the exchange protocol, too. The proof resembles the one for the clocks protocol; it requires a generalization of justified, called justified_with, to account for positive entries in every in-flight progress message M. The generalization has the same three disjuncts as justified, but relaxes the first and third disjunct to take into account an additional multiset N of justifying pointstamps. Usages of

locale graph =
  fixes weights :: ('vtx :: *finite*) ⇒ 'vtx ⇒ ('lbl :: {*order*, *monoid_add*}) *antichain*
  assumes (*l* :: 'lbl) ⩾ 0 **and** (*l*₁ :: 'lbl) ⩽ *l*₃ ⟶ *l*₂ ⩽ *l*₄ ⟶ *l*₁ + *l*₂ ⩽ *l*₃ + *l*₄
  and weights *l l* = {}
locale dataflow = graph summary
  for summary :: ('l :: *finite*) ⇒ 'l ⇒ ('sum :: {*order*, *monoid_add*}) *antichain* +
  fixes ⊕ :: ('t :: *order*) ⇒ 'sum ⇒ 't
  assumes *t* ⊕ 0 = *t* **and** (*t* ⊕ *s*) ⊕ *s*′ = *t* ⊕ (*s* + *s*′) **and** *t* ⩽ *t*′ ⟶ *s* ⩽ *s*′ ⟶ *t* ⊕ *s* ⩽ *t*′ ⊕ *s*′
  and path *l l xs* ⟶ *xs* ≠ [] ⟶ *t* < *t* ⊕ (∑ *xs*)

Figure 8.5: Locales for graphs and dataflows

justified_with instantiate N with the sum of messages following M in msg and the sender's temp.

definition justified_with :: 'p *zmset* ⇒ 'p *zmset* ⇒ 'p *zmset* ⇒ *bool* **where**
  justified_with *C M N* = (∀*p*. *M* #$_z$ *p* > 0 ⟶
    (∃*p*′ < *p*. *M* #$_z$ *p*′ < 0 ∨ *N* #$_z$ *p*′ < 0) ∨ (∃*p*′ < *p*. *C* #$_z$ *p*′ > 0) ∨ (*M* + *N*) #$_z$ *p* < *C* #$_z$ *p*)

We also derive the following additional property of glob, which shows that any in-flight progress updates to a pointstamp p, positive or negative, have a corresponding positive count for some pointstamp less or equal than p in the receiver's glob. We will use this property when combining in Section 8.8 the exchange protocol with the worker-local progress propagation, which we cover next in Section 8.7.

lemma *glob:* Spec *s* ⟶ alw (now (λ*c*. ∀*w w*′ *p*.
  (∃*M* ∈ set (msg *c w w*′). *p* ∈#$_z$ *M*) ⟶ (∃*p*′ ⩽ *p*. glob *c w*′ #$_z$ *p*′ > 0))) *s*

8.7 Locally Propagating Progress

The previous sections focused on the progress-relevant communication between workers and abstracted over the actual dataflow that is evaluated by each worker. Next, we refine this abstraction: we model the actual dataflow graph as a weighted directed graph with vertices representing operator input and output ports, termed *locations*. We do not distinguish between source and target locations and thus also not between internal and dataflow edges. Each weight denotes a minimum increment that is performed to a timestamp when it conceptually travels along the corresponding edge from one location to another. On a single worker, progress updates can be communicated locally, so that every operator learns which timestamps it may still receive in the future. We formalize Timely Dataflow's approach for this local communication: the algorithm gradually propagates learned pointstamp changes along dataflow edges to update downstream frontiers.

Figure 8.5 details our modeling of graphs and dataflows, which uses locales [Bal14] to capture our abstract assumptions on dataflows and timestamps. A locale lets us fix parameters (types and constants) and assume properties about them. In our setting, a weighted directed graph is given by a finite (class *finite*) type *'vtx* of vertices and a weights function that assigns each pair of vertices a weight. To express weights, we fix a type of labels *'lbl*, which we assume to be partially ordered (class *order*) and to form a monoid (class *monoid_add*) with the monoid operation $+$ and the neutral element 0. We assume that labels are non-negative and that $+$ on labels is monotone with respect to the partial order $\leqslant$. A weight is then an antichain of labels, that is a set of incomparable (with respect to $\leqslant$) labels, which we model as follows:

typedef $('t :: order)\ antichain = \{A :: 't\ set.\ \mathsf{finite}\ A \wedge (\forall a \in A.\ \forall b \in A.\ a \not\leqslant b \wedge b \not\leqslant a)\}$

We use standard set notation for antichains and omit type conversions from antichains to (signed) multisets. The empty antichain $\{\}$ is a valid weight, too, in which case we think of the involved vertices as not being connected to each other. Thus, the graph locale's final assumption expresses the non-existence of self-edges in a graph.

Within the graph locale, we can define the predicate $\mathsf{path} :: 'vtx \Rightarrow 'vtx \Rightarrow 'lbl\ list \Rightarrow bool$. Intuitively, $\mathsf{path}\ v\ w\ xs$ expresses that the list of

labels xs is a valid path from v to w (the empty list being a valid path only if $v = w$ and any weight $l \in \mathsf{weights}\ u\ v$ can extend a valid path from v to w to a path from u to w). We omit path's formal straightforward inductive definition. Note that even though self-edges are disallowed, cycles in graphs are possible (and desired). In other words, $\mathsf{path}\ v\ v\ xs$ can be true for a non-empty list xs.

The second locale, dataflow, has two purposes. First, it refines the generic graph terminology from vertices and labels to locations ($'l$) and summaries ($'sum$), which is the corresponding terminology used in Timely Dataflow. Second, it introduces the type for timestamps $'t$, which is partially ordered (class *order*) and an operation $\oplus$ (read as "results in") that applies a summary to a timestamp to obtain a new timestamp. We chose the asymmetric symbol for the operation to remind the reader that its two arguments have different types, timestamps and summaries. The locale requires the operation $\oplus$ to be well-behaved with respect to the available vocabulary on summaries (0, +, and $\leqslant$). Moreover, it requires that proper cycles xs have a path summary $\sum xs$ (defined by iterating +) that strictly increments any timestamp t.

Now, consider a function $P :: 'l \Rightarrow 't\ zmset$ that assigns each location a set of timestamps that it currently holds. We are interested in computing a lower bound of timestamps (with respect to the order $\leqslant$) that may arrive at any location for a given P. Timely Dataflow calls antichains that constitute such a lower bound frontiers. Formally, a frontier is the set of minimal incomparable elements that have a positive count in a signed multiset of timestamps.

definition antichain_of :: $'t\ set \Rightarrow 't\ set$ **where** antichain_of $A =$
$\{x \in A.\ \neg\exists y \in A.\ y < x\}$
lift_definition frontier :: $'t\ zmset \Rightarrow$
$'t\ antichain$ **is** $\lambda M.$ antichain_of $\{t.\ M\ \#_z\ t > 0\}$

Our frontier of interest, called the implied frontier, at location l can be computed directly for a given function P by adding, for every location l', every (minimal) possible path summary between l' and l, denoted by the antichain $\mathsf{path\_summary}\ l'\ l$, to every timestamp present at l' and computing the frontier of the result. Formally, we first lift $\oplus$ to signed multisets and antichains. Then, we use the lifted operator $\bigoplus$ to define the implied frontier.

definition $\bigoplus$ :: *'t zmset* ⇒ *'sum antichain* ⇒ *'t zmset* **where**
$M \bigoplus A = \sum_{s \in A} (\lambda t.\ t \oplus s)\ `\#_z\ M$
definition implied_frontier :: (*'l* ⇒ *'t zmset*) ⇒ *'l* ⇒
't antichain **where**
implied_frontier $P\ l$ =
frontier $(\sum_{l'} (\mathsf{pos}_z\ (P\ l') \bigoplus \mathsf{path\_summary}\ l'\ l))$

Above and elsewhere, given a signed multiset M, we write $f\ `\#_z\ M$ for the image (as a signed multiset) of f over M and $\mathsf{pos}_z\ M$ for the signed multiset of M's positive entries.

Computing the implied frontier for each location in this way (quadratic in the number of locations) would be too inefficient, especially because we want to frequently supply operators with up-to-date progress information. Instead, we follow the optimized approach implemented in Timely Dataflow: after performing some work and making some progress, operators start pushing relevant updates only to their immediate successors in the dataflow graph. The information gradually propagates and eventually converges to the implied frontier. Despite this local propagation not being a distributed protocol as such, we formalize it for a fixed dataflow in a similar state-machine style as the earlier exchange protocol.

Local propagation uses a configuration consisting of three signed multiset components.

record (*'l* :: *finite*, *'t* :: {*monoid_add*, *order*}) *conf* =
pts :: *'l* ⇒ *'t zmset*
imp :: *'l* ⇒ *'t zmset*
work :: *'l* ⇒ *'t zmset*

Following Timely Dataflow terminology, pointstamps pts are the present timestamps grouped by location (the P function from above). The implications imp are the output of the local propagation and contain an over-approximation of the implied frontier (as we will show). Finally, the worklist work is an auxiliary data structure to store not-yet propagated timestamps.

Initially, all implications are empty and worklists consist of the frontiers of the pointstamps.

definition Init :: (*'l*, *'t*) *conf* ⇒ *bool* **where**
Init $c = (\forall l.\ \mathsf{imp}\ c\ l = \{\#\}_z \wedge \mathsf{work}\ c\ l = \mathsf{frontier}\ (\mathsf{pts}\ c\ l))$

definition change_multiplicity :: $'l \Rightarrow 't \Rightarrow int \Rightarrow ('l, 't)\ conf \Rightarrow$
$('l, 't)\ conf \Rightarrow bool$ **where**
  change_multiplicity $l\ t\ n\ c\ c' = n \neq 0 \wedge (\exists t' \in$
frontier (implications $c\ l$). $t' \leqslant t) \wedge$
    $c' = c(\!|$pts $=$ (pts c)$(l :=$ pts $c\ l +$ replicate $n\ t)$,
      work $=$ (work c)$(l :=$ work $c\ l +$ frontier (pts $c'\ l) -$
frontier (pts $c\ l))|\!)$

definition propagate :: $'l \Rightarrow 't \Rightarrow ('l, 't)\ conf \Rightarrow ('l, 't)\ conf \Rightarrow$
$bool$ **where**
  propagate $l\ t\ c\ c' = t \in\#_z$ work $c\ l \wedge (\forall l'.\ \forall t' \in\#_z$ work $c\ l'.\ \neg t' < t) \wedge$
    $c' = c(\!|$imp $=$ (imp c)$(l :=$ imp $c\ l +$ replicate (work $c\ l\ \#_z\ t)\ t$,
      work $= \lambda l'.$ if $l = l'$ then $\{\#t' \in\#_z$ work $c\ l.\ t' \neq t\#\}$
        else work $c\ l' + ((\text{frontier (imp } c'\ l) -$
frontier (imp $c\ l)) \bigoplus$ summary $l\ l')|\!)$

definition Next :: $('l, 't)\ conf \Rightarrow ('l, 't)\ conf \Rightarrow bool$ **where**
  Next $c\ c' = (c = c') \vee (\exists l\ t\ n.$ change_multiplicity $l\ t\ n\ c\ c') \vee$
$(\exists l\ t.$ propagate $l\ t\ c\ c')$

Figure 8.6: Transition relation of the local progress propagation

The propagation proceeds by executing one of two actions shown in Figure 8.6. The action change_multiplicity constitutes the algorithm's information input: The system may have changed the multiplicity of some timestamp t at location l and can use this action to notify the propagation algorithm of the change. The change value n is required to be non-zero and the affected timestamp t must be witnessed by some timestamp in the implications. Note that the latter requirement prohibits executing this action in the initial state. The action updates the pointstamps according to the declared change. It also updates the worklist, but only if the update of the pointstamps affects the frontier of the pointstamps at l and moreover the worklists are updated merely by the change to the frontier.

The second action, propagate, applies the information for the timestamp t stored in the worklist at a given location l, to the location's implications (thus potentially enabling the first action). It also updates the worklists at the location's immediate successors in the dataflow graph. Again the worklist updates are filtered by whether they affect the frontier (of the implications) and are adjusted by the summary between l

and each successor. Importantly, only minimal timestamps (with respect to timestamps in worklists at all locations) may be propagated, which ensures that any timestamp will eventually disappear from all worklists.

The overall transition relation Next allows us to choose between these two actions and a stutter step. Together with Init, it gives rise to the predicate describing valid executions in the standard way: Spec s = now Init $s \wedge$ alw (relates Next) s.

We show that valid executions satisfy a safety invariant. Ideally, we would like to show that for any t with a positive count in pts at location l and for any path summary s between l and some location l', there is a timestamp in the (frontier of the) implications at l' that is less than or equal to $t \oplus s$. In other words, the location l' is aware that it may still encounter timestamp $t \oplus s$. Stated as above, the invariant does not hold, due to the not-yet-propagated progress information stored in the worklists. If some timestamp, however, does not occur in any worklist (formalized by the below work_vacant predicate), we obtain our desired invariant Safe.

definition work_vacant :: ('l, 't) *conf* ⇒ 't ⇒ *bool* **where**
  work_vacant $c\ t$ = ($\forall l\ l'\ s\ t'.\ t' \in\#_z$ work $c\ l \longrightarrow s \in$
path_summary $l\ l' \longrightarrow t' \oplus s \not\leqslant t$)
definition Safe :: ('l, 't) *conf stream* ⇒ *bool* **where**
  Safe c = ($\forall l\ l'\ t\ s.$ pts $c\ l \#_z t > 0 \wedge s \in$ path_summary $l\ l' \wedge$
work_vacant $c\ (t \oplus s) \longrightarrow$
    ($\exists t' \in$ frontier (imp $c\ l'$). $t' \leqslant t \oplus s$))
lemma *safe:* Spec $s \longrightarrow$ alw (now Safe) s

In our running WCC example, Safe is for example necessary to determine once operator b has received all incoming updates for a certain round of label propagation, which is encoded as a timestamp (t_1, t_2). If a pointstamp at port *b.3* was not correctly reflected in the frontier at *b.1* the operator may incorrectly determine that it has seen all incoming labels for a certain graph node and proceed to the next round of propagation. Safe states, that this cannot happen and all pointstamps are correctly reflected in relevant downstream frontiers.

The safety proof relies on two auxiliary invariants. First, implications have only positive entries. Second, the sum of the implication and the worklist at a given location l is equal to the sum of the frontier of the

pointstamps at l and the sum of all frontiers of the implications of all immediate predecessor locations l' (adjusted by the corresponding summary summary l' l).

While the above safety property is sufficient to prove safety of the combination of the local propagation and the exchange protocol in the next section, we also establish that the computed frontier of the implications converges to the implied frontier. Specifically, the two frontiers coincide for timestamps which are not contained in any of the worklists.

lemma *implied_frontier:* Spec $s \longrightarrow$
alw (now (λc. work_vacant $c\ t \longrightarrow$
($\forall l.\ t \in$ frontier (imp $c\ l$) $\longleftrightarrow\ t \in$ implied_frontier (pts c) l))) s

8.8 Progress Tracking

We are now ready to combine the two parts presented so far: the between-worker exchange of progress updates (Section 8.6) and the worker-local progress propagation (Section 8.7). The combined protocol takes pointstamp changes and determines per-location frontiers at each operator on each worker. It operates on configurations consisting of a single exchange protocol configuration (referred to with the prefix E) and for each worker a local propagation configuration (prefix P) and a Boolean flag indicating whether the worker has been properly initialized.

record ($'w$ :: *finite*, $'l$ :: *finite*, $'t$ :: {*monoid_add*, *order*}) *conf* =
exch :: ($'w$, $'l \times 't$) *E.conf*
prop :: $'w \Rightarrow$ ($'l$, $'t$) *P.conf*
init :: $'w \Rightarrow$ *bool*

As pointstamps in the exchange protocol, we use pairs of locations and timestamps. To order pointstamps, we use the following *could-result-in* relation, inspired by Abadi and Isard [AI15b].

definition $\leqslant_{cri}$ **where** $(l,t) \leqslant_{cri} (l',t') = (\exists s \in$
path_summary $l\ l'.\ t \oplus s \leqslant t')$

As required by the exchange protocol, this definition yields a partial order. In particular, antisymmetry follows from the assumption that proper cycles have a non-zero summary and transitivity relies on the operation $\oplus$ being monotone. Intuitively, $\leqslant_{cri}$ captures a notion of reachability in the

dataflow graph: as timestamp t traverses the graph starting at location l, it could arrive at location l', being incremented to timestamp t'. (In Timely Dataflow, an edge's summary represents the minimal increment to a timestamp when it traverses that edge.)

In an initial combined configuration, all workers are not initialized and all involved configurations are initial. Moreover, the local propagation's pointstamps coincide with exchange protocol's glob, which is kept invariant in the combined protocol.

definition Init :: $('w, 'l, 't)$ *conf* $\Rightarrow$ *bool* **where**
  Init c = $(\forall w.$ init c w = False$)$ $\wedge$ E.Init (exch c) $\wedge$
$(\forall w.$ P.Init (prop c w)$)$ $\wedge$
    $(\forall w$ l $t.$ P.pts (prop c w) l $\#_z$ t = E.glob (exch c) w $\#_z$ $(l, t))$

Figure 8.7 shows the combined protocol's transition relation Next. Most actions have identical names as the exchange protocol's actions and they mostly perform the corresponding actions on the exchange part of the configuration. In addition, the recv_upd action also performs several change_multiplicity local propagation actions: the receiver updates the state of its local propagation configuration for all received timestamp updates. The action propagate does not have a counterpart in the exchange protocol. It iterates, using the while_option combinator from Isabelle's library, propagation on a single worker until all worklists are empty. The term while_option b c s repeatedly applies c starting from the initial state s, until the predicate b is satisfied. Overall, it evaluates to Some s' satisfying $\neg b$ s' and $s' = c$ $(\cdots (c$ $s))$ with the least possible number of repetitions of c and to None if no such state exists. Thus, it is only possible to take the propagate action, if the repeated propagation terminates for the considered configuration. We believe that repeated propagation terminates for any configuration, but we do not prove this non-obvious[4] fact formally. Timely Dataflow also iterates propagation until all worklists of a worker become empty. This gives us additional empirical evidence that the iteration terminates on practical dataflows. Moreover, even if the it-

[4]Because propagation must operate on a globally minimal timestamp and because loops in the dataflow graph have a non-zero summary, repeated propagation will eventually forever remove any timestamp from any worklist. However, it is not as obvious why it eventually will stop introducing larger and larger timestamps in worklists. The termination argument must rely on the fact that only timestamps that modify the frontier of the implications are ever added to worklists.

definition recv_cap :: $'w \Rightarrow$ $'l \times$ $'t \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$
($'w$, $'l$, $'t$) $conf \Rightarrow$ $bool$ **where**
recv_cap w p c c' = E.recv_cap w p (exch c) (exch c') $\wedge$ prop c' =
prop c $\wedge$ init c' = init c

definition perf_op :: $'w \Rightarrow$ ($'l \times$ $'t$) $mset \Rightarrow$ ($'w \times$ ($'l \times$ $'t$)) $mset \Rightarrow$ ($'l \times$
$'t$) $mset \Rightarrow$
($'w$, $'l$, $'t$) $conf \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$ $bool$ **where**
perf_op w Δ_{neg} Δ_{data} Δ_{self} c c' =
E.perf_op w Δ_{neg} Δ_{data} Δ_{self} (exch c) (exch c') $\wedge$
prop c' = prop c $\wedge$ init c' = init c

definition send_upd :: $'w \Rightarrow$ ($'l \times$ $'t$) $set \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$
($'w$, $'l$, $'t$) $conf \Rightarrow$ $bool$ **where**
send_upd w P c c' = E.send_upd (exch c) (exch c') w P $\wedge$ prop c' =
prop c $\wedge$ init c' = init c

definition cm_all :: ($'l$, $'t$) $P.conf \Rightarrow$ ($'l \times$ $'t$) $zmset \Rightarrow$
($'l$, $'t$) $P.conf$ **where**
cm_all c Δ =
Set.fold ($\lambda(l,t)$ c. SOME c'. P.change_multiplicity c c' l t (Δ $\#_z$ (l,t))) c
{(l,t). $(l,t) \in \#_z$ Δ}

definition recv_upd :: $'w \Rightarrow$ $'w \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$
$bool$ **where**
recv_upd w w' c c' = init c w' $\wedge$ E.recv_upd w t (exch c) (exch c') $\wedge$
prop c' = (prop c)(w' := cm_all (prop c w') (hd (E.msg (exch c)))) $\wedge$
init c' = init c

definition propagate :: $'w \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$
$bool$ **where**
propagate w c c' = exch c' = exch c $\wedge$ init c' = (init c)(w := True) $\wedge$
(Some $\circ$ prop c') = (Some $\circ$ prop c)(w := while_option
(λc. $\exists l$. P.work c $l \neq$
{#}$_z$) (λc. SOME c'. $\exists l$ t. P.propagate l t c c') (prop c w))

definition Next :: ($'w$, $'l$, $'t$) $conf \Rightarrow$ ($'w$, $'l$, $'t$) $conf \Rightarrow$ $bool$ **where**
Next c c' = $(c = c') \vee$ ($\exists w$ p. recv_cap w p c c') $\vee$
($\exists w$ Δ_{neg} Δ_{data} Δ_{self}. perf_op w Δ_{neg} Δ_{data} Δ_{self} c c') $\vee$
($\exists w$ P. send_upd w P c c') $\vee$ ($\exists w$ w'. recv_upd w w' c c') $\vee$
($\exists w$. propagate w c c')

Figure 8.7: Transition relation of the combined progress tracker

eration were to not terminate for some worker on some dataflow (both in Timely Dataflow and in our model), our combined protocol can faithfully capture this behavior by not executing the propagate action, but also not any other action involving the looping worker, thus retaining safety for the rest of the workers. Finally, any worker that has completed at least one propagation action is considered to be initialized (by setting its init flag to True).

The Init predicate and the Next relation give rise to the familiar specification of valid executions Spec s = now Init $s \wedge$ alw (relates Next) s. Safety of the combined protocol can be described informally as follows: Every initialized worker w has some evidence for the existence of a timestamp t at location l at *any* worker w' in the frontier of its (i.e., w's) implications at all locations l' reachable from l. Formally, E.rec contains the timestamps that exist in the system:

definition Safe :: ('w, 'l, 't) *conf stream* ⇒ *bool* **where**
  Safe c = ($\forall w\ l\ l'\ t\ s$. init $c\ w \wedge$ E.rec (exch c) $\#_z$ $(l, t) >$
$0 \wedge s \in$ path_summary $l\ l' \longrightarrow$
    ($\exists t' \in$ frontier (P.imp (prop $c\ w$) l'). $t' \leqslant t \oplus s$)

Our main formalized result is the statement that the above predicate is an invariant.

lemma *safe:* Spec $s \longrightarrow$ alw (now Safe) s

The proof proceeds by lifting (and then combining) the safety statements and some auxiliary invariants of the exchange protocol and the local propagation to the combined execution. The lifting step is feasible, because we included stutter steps in the modeling of these components. In particular, the projection of a valid execution to the exchange configurations results in a valid execution of the exchange protocol: the propagate step constitutes a stutter step for the exchange configuration. In contrast, the projection to the local propagation configuration does not result in a valid execution of the local propagation, but in an execution that takes steps according to the reflexive transitive closure of the local propagation's transition relation P.Next: the steps propagate and recv_upd can take an arbitrary number of local propagation steps (whereas other transitions stutter from the point of view of local propagation). Fortunately, safety properties are easy to lift to such "big-step" executions.

In the combined progress tracking protocol, safety guarantees that if a pointstamp is present at an operator's port, it is correctly reflected at every downstream port. In the WCC example, when deployed on two workers, each operator is instantiated twice, once on each worker. If a pointstamp (*b.3*, `(3,0)`) is present on port *b.3* of one of the instances of operator *b*, the frontier at *c.1* on all workers must contain a t such that $t \preceq (3, 0)$. Due to the summary between *c.1* and *c.2*, frontiers at *c.2* and *b.1* must contain a t such that $t \preceq (3, 1)$. As an example, this ensures that operator *b* waits for each of its instances to complete the first round propagation of all labels before it chooses the lowest label for the next round.

8.9 Conclusions

We have presented an Isabelle/HOL formalization of Timely Dataflow's progress tracking protocol, including the verification of its safety. Compared to an earlier formalization by Abadi et al. [Aba+13], our protocol is both more general, which allows it to capture behaviors present in the implementations of Timely Dataflow and absent in Abadi et al.'s model, and more detailed in that it explicitly models the local propagation of progress information.

Our formalization spans about 7 000 lines of Isabelle definitions and proofs. These are roughly distributed as follows over the components we presented: basic properties of graphs and signed multisets (1 000), exchange protocol (3 100), local propagation (1 700), combined protocol (1 200). This is comparable in size to the TLA$^+$ Proof System formalization by Abadi et al., even though we formalized a significantly more detailed, complex, and realistic variant of the progress tracking protocol. Ground to this claim is the fact that we had actually started our formalization by porting significant parts of the TLA$^+$ Proof System formalization to Isabelle. We completed the proofs of their two main safety statement within one person-week in about 1 000 lines of Isabelle (not included above). Our use of Isabelle's library for linear temporal logic on streams (in particular, the coinductive predicate alw) allowed us to copy directly a vast majority of the TLA$^+$ definitions. Additionally, Isabelle's mature proof automation allowed us to apply a fairly mechanical porting process to many of the proofs. Most ported lemmas could be proved either

directly by Sledgehammer [PB10] or by sketching an Isar [Wen07] proof skeleton of the main proof steps and discharging most of the resulting subgoals with Sledgehammer.

In the subsequent development of the combined protocol, Isabelle's locales [Bal14] were an important asset. By confining the exchange protocol and the local propagation each to their own local assumptions, we were able to develop them in parallel and in their full generality. Thus, we obtain formal models not only of the combined protocol itself but also of these two subsystems in a generality that goes beyond what is needed for the concrete combined instance. For example, although the combined protocol uses the could-result-in order, the exchange protocol works for any partial order on pointstamps. Moreover, the combined protocol always propagates until all worklists are empty, even though the local propagation's safety supports small-step propagation, resulting in a more fine-grained safety property via work_vacant.

In our formalization, we make extensive use of signed multisets [BFT17]. The alternative (used in the TLA$^+$ Proof System formalization), would be to use integer-valued functions instead. The signed multiset type additionally captures a finite domain assumption, which it was convenient not to carry around explicitly and in particular simplified reasoning about summations. The expected downside of having separate types for function-like (*mset*) and set-like (*antichain*) objects was the need to insert explicit type conversions and to transfer properties across these conversions. Both complications were to some extent alleviated by Lifting and Transfer [HK13].

9 Conclusion

This thesis argues that a well-considered abstraction of the underlying coordination mechanism of a distributed dataflow streaming system enables the construction of higher-level protocols for dynamic scaling, index sharing, fault tolerance, and flow control without modifications to the core system.

In Chapter 4 I introduced timestamp tokens, a coordination primitive for dataflow systems. Timestamp tokens decouple the sophistication of the operators' own scheduling logic from the system's coordination mechanism. Libraries built with timestamp tokens can write operator abstractions with more sophisticated coordination logic by encoding it through timestamp tokens. The following chapters demonstrate this approach with libraries that address various dataflow system challenges such as index sharing, flow control, dynamic scaling, and fault tolerance.

9.1 Composing a system from libraries

Using timestamp tokens, each new mechanism is built as a library, without changes to the underlying system. An unresolved question in this thesis is how to ensure these libraries compose when multiple mechanisms are needed in a specific deployment.

Other systems like Flink and Spark Streaming expose a very abstract operator interface to allow the system to implement mechanisms such as rescaling and fault-tolerance transparently to the dataflow program. A similar approach may aid composability in systems built on timestamp tokens: for example, the fault-tolerance and rescaling libraries can expose a common, very simple tuple-at-a-time programming interface; however

this choice requires selecting an API that enables each of the library features, even when they are unneeded. This often comes at a performance cost.

A more efficient, but probably more engineering-intensive approach is for libraries to provide APIs that enable the client program to extract maximum performance; in this case libraries higher in the stack need to be compatible with both the lowest level timestamp token programming interface and one of the higher-level interfaces provided by the other libraries lower in the stack. Stacking these libraries is necessary to achieve multiple system goals, e.g. for a system that supports both index sharing and fault tolerance. Choosing the order the libraries would appear in the stack is also an important consideration in terms of API design and its performance cost.

9.2 You may not need synchronization

Chapters 5, 6, and 7 presented mechanisms for efficiently managing dataflow systems concerns. These are all built on timestamp tokens, but they also independently address system issues where existing techniques were absent or inadequate for modern, high-efficiency dataflow systems.

They all rely on carefully selecting and minimizing the synchronization points necessary for the correct functioning of the system and for generating correct results. Asynchornous coordination and synchronization rely on the underlying Timely Dataflow progress tracking protocol, and intract with it via the timestamp tokens-based API.

Instead of a global synchronization signal, Megaphone (Chapter 5) uses precise per-operator coordination signals based on timestamps that precisely sequence re-partitioning operations and data-processing to ensure that data is routed to and processed by operators where the relevant state resides.

Similarly, Shared Arrangements (Chapter 6) do not require locks on shared indicies because they can rely on timestamps and coordination signals from timestamp tokens to perform independent reads and writes to the single-threaded, multi-versioned Arrangement data structure: consumers of the index can subset their view of the indexed data by time independently from each other, without locks or explicit ordering of writes and reads.

The fault tolerance mechanism discussed in Chapter 7, CL, removes synchronization from the critical path of data processing by relying on timestamp information to record state changes. timestamp tokens enabled the construction an asynchronous garbage collection protocol for recorded state using the existing coordination signalling mechanism in timely dataflow. Explicit synchronization only happens on recovery, when operators need to negotiate a safe recovery point.

Minimizing critical-path synchronization is critical in building efficient concurrent systems as it allows each processor to make progress independently and avoids idling while waiting for synchronization signals to arrive. By minimizing critical-path synchronization the systems mechanisms discussed in this dissertation can be adapted to cope with the more complex execution model, while introducing minimal overhead, to avoid squandering the increased efficiency of modern data-parallel dataflow systems.

List of Tables

List of Figures

Bibliography

[A F73] Edward A. Feustel. "On The Advantages of Tagged Architecture". In: *Computers, IEEE Transactions on* C-22 (Aug. 1973), pp. 644–656. DOI: 10.1109/TC.1973.5009130 (cit. on p. 46).

[Aba+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Aurora: a new model and architecture for data stream management". In: *The VLDB Journal* 12.2 (Aug. 2003), pp. 120–139. ISSN: 0949-877X. DOI: 10.1007/s00778-003-0095-z. URL: https://doi.org/10.1007/s00778-003-0095-z (cit. on pp. 1, 17, 28).

[Aba+05] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. "The Design of the Borealis Stream Processing Engine." In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289 (cit. on p. 28).

[Aba+13] Martín Abadi, Frank McSherry, Derek G. Murray, and Thomas L. Rodeheffer. "Formal Analysis of a Distributed Algorithm for Tracking Progress". In: *Lecture Notes in Computer Science* (2013), pp. 5–19. ISSN: 1611-3349. DOI: 10.1007/978-3-642-38592-6_2. URL: http://dx.doi.org/10.1007/978-3-642-38592-6_2 (cit. on pp. 134, 135, 140–142, 145, 162).

[Ahm+12] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". In: *Proc. VLDB Endow.* 5.10 (2012), pp. 968–979. DOI: 10.14778/2336664.2336670. URL: http://vldb.org/pvldb/vol5/p968%5C_yanifahmad%5C_vldb2012.pdf (cit. on pp. 74, 80).

[AI15a] Martín Abadi and Michael Isard. "Timely Dataflow: A Model". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Susanne Graf and Mahesh Viswanathan. Cham: Springer International Publishing, 2015, pp. 131–145. ISBN: 978-3-319-19195-9 (cit. on p. 136).

[AI15b] Martín Abadi and Michael Isard. "Timely Dataflow: A Model". In: *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*. Vol. 9039. LNCS. Springer, 2015, pp. 131–145. DOI: 10.1007/978-3-319-19195-9_9. URL: https://doi.org/10.1007/978-3-319-19195-9%5C_9 (cit. on p. 158).

[Aki+13] Tyler Akidau, Sam Whittle, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, and Paul Nordstrom. "MillWheel: Fault-Tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044. DOI: 10.14778/2536222.2536229 (cit. on p. 43).

[Aki+15] Tyler Akidau, Eric Schmidt, Sam Whittle, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, and Frances Perry. "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-Of-Order Data Processing". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1792–1803. DOI: 10.14778/2824032.2824076 (cit. on pp. 15, 33).

[Ama] Amazon. *What is Amazon S3?* https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html. Accessed 2022-10-19 (cit. on p. 122).

[AMP15] Martín Abadi, Frank McSherry, and Gordon D. Plotkin. "Foundations of Differential Dataflow". In: *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Andrew M. Pitts. Vol. 9034. Lecture Notes in Computer Science. Springer, 2015, pp. 71–83. DOI: 10.1007/978-3-662-46678-0_5. URL: https://doi.org/10.1007/978-3-662-46678-0%5C_5 (cit. on p. 82).

[Apa] Apache Storm developers. *Apache Storm*. https://storm.apache.org. Accessed 2022-10-19 (cit. on pp. 15, 33, 43).

[Ara+04] 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*. Technical Report 2004-20. Stanford InfoLab, 2004. URL: http://ilpubs.stanford.edu:8090/641/ (cit. on p. 28).

[Bal+05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. "Fault-tolerance in the Borealis Distributed Stream Processing System". In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 13–24. ISBN: 1-59593-060-4. DOI:

10.1145/1066157.1066160. URL: http://doi.acm.org/10.1145/1066157.1066160 (cit. on p. 1).

[Bal14] Clemens Ballarin. "Locales: A Module System for Mathematical Theories". In: *J. Autom. Reason.* 52.2 (2014), pp. 123–153. DOI: 10.1007/s10817-013-9284-7. URL: https://doi.org/10.1007/s10817-013-9284-7 (cit. on pp. 153, 163).

[Ban+86] François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. "Magic Sets and Other Strange Ways to Implement Logic Programs". In: *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*. Ed. by Avi Silberschatz. ACM, 1986, pp. 1–15. DOI: 10.1145/6012.15399. URL: https://doi.org/10.1145/6012.15399 (cit. on p. 105).

[BC19] Véronique Benzaken and Evelyne Contejean. "A Coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 249–261. DOI: 10.1145/3293880.3294107. URL: https://doi.org/10.1145/3293880.3294107 (cit. on p. 136).

[Ben+18] Véronique Benzaken, Evelyne Contejean, Chantal Keller, and E. Martins. "A Coq Formalisation of SQL's Execution Engines". In: *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*. 2018, pp. 88–107. DOI: 10.1007/978-3-319-94821-8_6. URL: https://doi.org/10.1007/978-3-319-94821-8%5C_6 (cit. on p. 136).

[BFT17] Jasmin Christian Blanchette, Mathias Fleury, and Dmitriy Traytel. "Nested Multisets, Hereditary Multisets, and Syntactic Ordinals in Isabelle/HOL". In: *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*. Vol. 84. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 11:1–11:18. DOI: 10.4230/LIPIcs.FSCD.2017.11. URL: https://doi.org/10.4230/LIPIcs.FSCD.2017.11 (cit. on pp. 140, 163).

[Bru+21] Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. "Verified Progress Tracking for Timely Dataflow". In: *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 10:1–10:20. DOI: 10.4230/LIPIcs.ITP.2021.10. URL: https://doi.org/10.4230/LIPIcs.ITP.2021.10 (cit. on pp. 7–9, 33, 133).

[BW01] Shivnath Babu and Jennifer Widom. “Continuous Queries over Data Streams”. In: *SIGMOD Rec.* 30.3 (2001), pp. 109–120. DOI: 10.1145/603867.603884. URL: https://doi.org/10.1145/603867.603884 (cit. on pp. 28, 80).

[Car+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. “Apache flink: Stream and batch processing in a single engine”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cit. on pp. 2, 15, 33, 41, 43, 74, 78).

[Car+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. “State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing”. In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1718–1729. DOI: 10.14778/3137765.3137777 (cit. on p. 121).

[Çet+16] Uğur Çetintemel, Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Mitch Cherniack, Jeong-Hyon Hwang, Samuel Madden, Anurag Maskey, Alexander Rasin, et al. “The Aurora and Borealis Stream Processing Engines”. In: *Data Stream Management*. Springer, 2016, pp. 337–359 (cit. on p. 17).

[CGM09] Badrish Chandramouli, Jonathan Goldstein, and David Maier. “On-the-fly Progress Detection in Iterative Stream Queries”. In: *Proc. VLDB Endow.* 2.1 (2009), pp. 241–252. DOI: 10.14778/1687627.1687655. URL: http://www.vldb.org/pvldb/vol2/vldb09-224.pdf (cit. on p. 43).

[Cha+10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. “Verifying Safety Properties with the TLA+ Proof System”. In: *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*. Ed. by Jürgen Giesl and Reiner Hähnle. Vol. 6173. Lecture Notes in Computer Science. Springer, 2010, pp. 142–148. DOI: 10.1007/978-3-642-14203-1_12. URL: https://doi.org/10.1007/978-3-642-14203-1%5C_12 (cit. on p. 136).

[Cha03] Sirish Chandrasekaran. *TelegraphCQ: Continuous Dataflow Processing for an Uncertain World.; CIDR*. 2003. URL: http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf (cit. on p. 80).

[Che+00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. “NiagaraCQ: A Scalable Continuous Query System for Internet Databases”. In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. Ed. by Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein. ACM, 2000, pp. 379–390. DOI: 10.1145/342009.335432. URL: https://doi.org/10.1145/342009.335432 (cit. on p. 99).

[Chi+15] David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. "Beyond the PDP-11: Architectural Support for a Memory-Safe C Abstract Machine". In: *SIGPLAN Not.* 50.4 (Mar. 2015), pp. 117–130. ISSN: 0362-1340. DOI: 10.1145/2775054.2694367. URL: http://doi.acm.org/10.1145/2775054.2694367 (cit. on p. 46).

[Chu+17] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. "Cosette: An Automated Prover for SQL". In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 2017. URL: http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf (cit. on p. 136).

[CJ75] Ellis Cohen and David Jefferson. "Protection in the Hydra Operating System". In: *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*. SOSP '75. Austin, Texas, USA: ACM, 1975, pp. 141–160. DOI: 10.1145/800213.806532. URL: http://doi.acm.org/10.1145/800213.806532 (cit. on p. 46).

[CL85] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pp. 63–75. ISSN: 0734-2071. DOI: 10.1145/214451.214456. URL: http://doi.acm.org/10.1145/214451.214456 (cit. on p. 118).

[CPV09] George Candea, Neoklis Polyzotis, and Radek Vingralek. "A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses". In: *Proc. VLDB Endow.* 2.1 (2009), pp. 277–288. DOI: 10.14778/1687627.1687659. URL: http://www.vldb.org/pvldb/vol2/vldb09-553.pdf (cit. on p. 80).

[Dar19] Erik Darling. *Locks Taken During Indexed View Modifications*. https://www.brentozar.com/archive/2018/09/locks-taken-during-indexed-view-modifications/. Accessed: 2022-10-03. Sept. 2019 (cit. on p. 78).

[DB13] Jeffrey Dean and Luiz André Barroso. "The tail at scale". In: *Commun. ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794 (cit. on p. 2).

[Dec20] Sára Decova. "Modelling and Verification of the Timely Dataflow Progress Tracking Protocol". en. Master Thesis. Zurich: ETH Zurich, 2020. DOI: 10.3929/ethz-b-000444762 (cit. on pp. 8, 9, 133).

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. Ed. by Eric A. Brewer and Peter Chen. USENIX Association, 2004, pp. 137–150. URL: http://www.usenix.org/events/osdi04/tech/dean.html (cit. on pp. 1, 15, 78).

[DLV] DLVSYSTEM authors. *DLVSYSTEM*. http://www.dlvsystem.com. Accessed 2022-10-19 (cit. on p. 105).

[DOT20] Tomás Díaz, Federico Olmedo, and Éric Tanter. "A mechanized formalization of GraphQL". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*. Ed. by Jasmin Blanchette and Catalin Hritcu. ACM, 2020, pp. 201–214. DOI: 10.1145/3372885.3373822. URL: https://doi.org/10.1145/3372885.3373822 (cit. on p. 136).

[Dup+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19 (cit. on p. 58).

[DV66] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: http://doi.acm.org/10.1145/365230.365252 (cit. on p. 46).

[eBa] eBay. *eBay*. https://www.ebay.com. Accessed 2022-10-19 (cit. on p. 13).

[Ess+18] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. "Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data". In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 799–815. URL: https://www.usenix.org/conference/osdi18/presentation/essertel (cit. on pp. 113, 114).

[Fab74] R. S. Fabry. "Capability-based Addressing". In: *Commun. ACM* 17.7 (July 1974), pp. 403–412. ISSN: 0001-0782. DOI: 10.1145/361011.361070. URL: http://doi.acm.org/10.1145/361011.361070 (cit. on p. 46).

[Fan+19] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. "Scaling-Up In-Memory Datalog Processing: Observations and Techniques". In: *Proc. VLDB Endow.* 12.6 (2019), pp. 695–708. DOI: 10.14778/3311880.3311886. URL: http://www.vldb.org/pvldb/vol12/p695-fan.pdf (cit. on pp. 108–111).

[GAK12] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. "SharedDB: Killing One Thousand Queries With One Stone". In: *Proc. VLDB Endow.* 5.6 (2012), pp. 526–537. DOI: 10.14778/2168651.2168654. URL: http://vldb.org/pvldb/vol5/p526%5C_georgiosgiannikis%5C_vldb2012.pdf (cit. on p. 80).

[Gje+18] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. "Noria: dynamic, partially-stateful data-flow for high-performance web applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/gjengset (cit. on pp. 74, 78, 80).

[Gog+15] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek Gordon Murray, Steven Hand, and Michael Isard. "Broom: Sweeping Out Garbage Collection from Big Data Systems". In: *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. Ed. by George Candea. USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog (cit. on p. 104).

[Gom+17] Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. "Verifying strong eventual consistency in distributed systems". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017), 109:1–109:28. DOI: 10.1145/3133933. URL: https://doi.org/10.1145/3133933 (cit. on p. 136).

[Gon+14] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 599–613. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez (cit. on p. 111).

[Gra94] G. Graefe. "Volcano – An extensible and parallel query evaluation system". In: *IEEE Transactions on Knowledge and Data Engineering* 6.1 (1994), pp. 120–135. DOI: 10.1109/69.273032 (cit. on pp. 1, 18–21, 25).

[Gu+19] Jiaqi Gu, Yugo H. Watanabe, William A. Mazza, Alexander Shkapsky, Mohan Yang, Ling Ding, and Carlo Zaniolo. "RaSQL: Greater Power and Performance for Big Data Analytics with Recursive-aggregate-SQL on Spark". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 467–484. DOI: 10.1145/3299869.3324959.

URL: https://doi.org/10.1145/3299869.3324959 (cit. on p. 111).

[Gun+10] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. Ed. by Remzi H. Arpaci-Dusseau and Brad Chen. USENIX Association, 2010, pp. 75–88. URL: http://www.usenix.org/events/osdi10/tech/full%5C_papers/Gunda.pdf (cit. on pp. 78, 80).

[Han+20] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. "Storage Systems are Distributed Systems (So Verify Them That Way!)" In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 99–115. URL: https://www.usenix.org/conference/osdi20/presentation/hance (cit. on p. 136).

[Haw+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 1–17. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815428. URL: http://doi.acm.org/10.1145/2815400.2815428 (cit. on p. 136).

[HBK20] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris: session-type based reasoning in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 6:1–6:30. DOI: 10.1145/3371074. URL: https://doi.org/10.1145/3371074 (cit. on p. 136).

[HK13] Brian Huffman and Ondrej Kuncar. "Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL". In: *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Vol. 8307. LNCS. Springer, 2013, pp. 131–146. DOI: 10.1007/978-3-319-03545-1_9. URL: https://doi.org/10.1007/978-3-319-03545-1%5C_9 (cit. on p. 163).

[Hoa78] C. A. R. Hoare. "Communicating Sequential Processes". In: *Commun. ACM* 21.8 (1978), pp. 666–677. DOI: 10.1145/359576.359585. URL: https://doi.org/10.1145/359576.359585 (cit. on p. 136).

[Hof+18] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. "Megaphone: Live state migration for distributed streaming dataflows". In: *CoRR* abs/1812.01371 (2018). arXiv: 1812.01371. URL: http://arxiv.org/abs/1812.01371 (cit. on p. 130).

[Hof+19] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. "Megaphone: Latency-conscious state migration for distributed streaming dataflows". In: *Proc. VLDB Endow.* 12.9 (2019), pp. 1002–1015. DOI: 10.14778/3329772.3329777. URL: http://www.vldb.org/pvldb/vol12/p1002-hoffmann.pdf (cit. on pp. 9, 44, 66, 70, 71).

[Hof+20] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, and Lorenzo Selvatici. *GitHub: strymon-system / megaphone*. https://github.com/strymon-system/megaphone. Accessed 2021-12-14. 2020 (cit. on pp. 66, 127).

[Hof19] Moritz Hoffmann. "Managing and understanding distributed stream processing". en. Doctoral Thesis. Zurich: ETH Zurich, 2019. DOI: 10.3929/ethz-b-000378081 (cit. on pp. 9, 70).

[Isa+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 - EuroSys '07* (2007). DOI: 10.1145/1272996.1273005. URL: http://dx.doi.org/10.1145/1272996.1273005 (cit. on pp. 1, 15, 78).

[Jem] Jemalloc developers. *Jemalloc memory allocator*. http://jemalloc.net. Accessed 2022-10-19 (cit. on p. 95).

[Jun+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151. URL: https://doi.org/10.1017/S0956796818000151 (cit. on p. 136).

[KRM19] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. "AStream: Ad-hoc Shared Stream Processing". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, 2019, pp. 607–622. DOI: 10.1145/3299869.3319884. URL: https://doi.org/10.1145/3299869.3319884 (cit. on p. 80).

[Lam02] Leslie Lamport. "Paxos Made Simple, Fast, and Byzantine". In: *Proceedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*. Ed. by Alain Bui and Hacène Fouchal. Vol. 3. Studia Informatica Universalis. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9 (cit. on p. 134).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563 (cit. on p. 28).

[Lat16] Andrea Lattuada. "Programmable scheduling in a stream processing system". de. Masterarbeit. Systems Group, Department of Computer Science, ETH Zurich. 2015-2016. Nr. 144. Master Thesis. Zürich: ETH Zurich, 2016. DOI: 10.3929/ethz-a-010648300 (cit. on pp. 7, 8, 43, 44, 69).

[LBC16] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. "Chapar: certified causally consistent distributed key-value stores". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 2016, pp. 357–370. DOI: 10.1145/2837614.2837622. URL: https://doi.org/10.1145/2837614.2837622 (cit. on p. 136).

[LM22] Andrea Lattuada and Frank McSherry. *Timestamp tokens: a better coordination primitive for data-processing systems*. 2022. DOI: 10.48550/ARXIV.2210.06113. URL: https://arxiv.org/abs/2210.06113 (cit. on pp. 7, 8, 33, 43, 69).

[LMC16] Andrea Lattuada, Frank McSherry, and Zaheer 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 - BeyondMR '16*. ACM Press, 2016. DOI: 10.1145/2926534.2926544 (cit. on pp. 44, 69).

[Mal+10] J. Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. "Toward a verified relational database management system". In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 2010, pp. 237–248. DOI: 10.1145/1706299.1706329. URL: https://doi.org/10.1145/1706299.1706329 (cit. on p. 136).

[Mar19] Lorenzo Martini. "Performance implications of a Dataflow System's Communication Plane". en. Master Thesis. Zurich: ETH Zurich, 2019. DOI: 10.3929/ethz-b-000338594 (cit. on p. 8).

[Mat] Materialize authors. *Materialize: Incrementally-updated materialized views*. https://materialize.com. Accessed 2022-10-19. URL: https://materialize.com (cit. on p. 135).

[McS+13] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of CIDR 2013*. Jan. 2013. URL: https://www.microsoft.com/en-us/research/publication/differential-dataflow/ (cit. on pp. 17, 44, 73, 135).

[McS+18] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. *Shared Arrangements: practical inter-query sharing for streaming dataflows*. 2018. DOI: 10.48550/ARXIV.1812.02639. URL: https://arxiv.org/abs/1812.02639 (cit. on pp. 8, 73).

[McS+20] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. "Shared Arrangements: practical inter-query sharing for streaming dataflows". In: *Proc. VLDB Endow.* 13.10 (2020), pp. 1793–1806. URL: http://www.vldb.org/pvldb/vol13/p1793-mcsherry.pdf (cit. on pp. 8, 33, 70, 73, 118).

[MD] Frank McSherry and Differential dataflow contributors. *Differential dataflow*. https://github.com/TimelyDataflow/differential-dataflow. Accessed 2022-10-19 (cit. on pp. 73, 77, 82, 94).

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. "Scalability! But at what COST?" In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry (cit. on p. 1).

[MK14] Nicholas D. Matsakis and Felix S. Klock II. "The Rust Language". In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*. HILT '14. Portland, Oregon, USA: ACM, 2014, pp. 103–104. ISBN: 978-1-4503-3217-0. DOI: 10.1145/2663171.2663188. URL: http://doi.acm.org/10.1145/2663171.2663188 (cit. on p. 48).

[MPI] MPI Forum. *MPI Forum, MPI-3.1*. https://www.mpi-forum.org/docs/. Accessed: 2021-12-14. URL: https://www.mpi-forum.org/docs/ (cit. on p. 43).

[MT] Frank McSherry and Timely Dataflow contributors. *Timely Dataflow*. https://github.com/TimelyDataflow/timely-dataflow. Accessed 2022-10-19 (cit. on pp. 15, 33, 34, 42, 48, 94, 119, 127, 132–136).

[Mul+90] S. J. Mullender, G. van Rossum, A. S. Tananbaum, R. van Renesse, and H. van Staveren. "Amoeba: a distributed operating system for the 1990s". In: *Computer* 23.5 (May 1990), pp. 44–53. ISSN: 0018-9162. DOI: 10.1109/2.53354 (cit. on p. 46).

[Mur+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. ACM Press, 2013. DOI: 10.1145/2517349.2522738 (cit. on pp. 2, 15, 33, 34, 38, 43, 73, 74, 78, 88, 134–136).

[Mur+16] Derek Gordon Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martín Abadi. "Incremental, iterative data processing with timely dataflow". In: *Commun. ACM* 59.10 (2016), pp. 75–83. DOI: 10.1145/2983551. URL: https://doi.org/10.1145/2983551 (cit. on pp. 134, 135).

[NDK16] Milos Nikolic, Mohammad Dashti, and Christoph Koch. "How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance With Batch Updates". In: *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, 2016. DOI: 10.1145/2882903.2915246 (cit. on pp. 76, 95, 96, 113).

[OO14] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro (cit. on p. 134).

[Pac+17] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. "Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications". In: *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*. Ed. by Peter A. Boncz and Josep Lluís Larriba-Pey. ACM, 2017, 12:1–12:7. DOI: 10.1145/3078447.3078459. URL: https://doi.org/10.1145/3078447.3078459 (cit. on pp. 99–101).

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. "Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers". In: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*. Vol. 2. EPiC Series in Computing. EasyChair, 2010, pp. 1–11. DOI: 10.29007/36dt. URL: https://doi.org/10.29007/36dt (cit. on p. 163).

[Pos] PostgreSQL Global Development Group. *The PostgreSQL Database Management System*. https://www.postgresql.org. Accessed 2022-10-19 (cit. on p. 78).

[RB19] Leonid Ryzhyk and Mihai Budiu. "Differential Datalog". In: *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019*. Ed. by Mario Alviano and Andreas Pieris. Vol. 2368. CEUR Workshop Proceedings. CEUR-WS.org, 2019, pp. 56–67. URL: http://ceur-ws.org/Vol-2368/paper6.pdf (cit. on p. 77).

[Rus] Rust developers and contributors. *Rust programming language*. https://www.rust-lang.org. Accessed: 2021-12-14 (cit. on p. 48).

[Sel04] Lorenzo Selvatici. "A Streaming System with Coordination-Free Fault-Tolerance". en. MA thesis. Zurich: ETH Zurich, 2020-04. DOI: 10.3929/ethz-b-000455359 (cit. on pp. 8, 117).

[SGL15] Jiwon Seo, Stephen Guo, and Monica S. Lam. "SociaLite: An Efficient Graph Query Language Based on Datalog". In: *IEEE Trans. Knowl. Data Eng.* 27.7 (2015), pp. 1824–1837. DOI: 10.1109/TKDE.2015.2405562. URL: https://doi.org/10.1109/TKDE.2015.2405562 (cit. on pp. 105, 111).

[Shk+16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. "Big Data Analytics With Datalog Queries on Spark". In: *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*. ACM Press, 2016. DOI: 10.1145/2882903.2915229 (cit. on pp. 105, 106, 110, 111).

[Spr+20] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. "Igloo: soundly linking compositional refinement and separation logic for distributed system verification". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 152:1–152:31. DOI: 10.1145/3428220. URL: https://doi.org/10.1145/3428220 (cit. on p. 136).

[Str95] Bjarne Stroustrup. *The design and evolution of C++*. Addison-Wesley, 1995. ISBN: 978-0-201-54330-8 (cit. on p. 4).

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and proving with distributed protocols". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 28:1–28:30. DOI: 10.1145/3158116. URL: https://doi.org/10.1145/3158116 (cit. on p. 136).

[Ter+92] Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. "Continuous Queries over Append-Only Databases". In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*. Ed. by Michael Stonebraker. ACM Press, 1992, pp. 321–330. DOI: 10.1145/130283.130333. URL: https://doi.org/10.1145/130283.130333 (cit. on p. 28).

[TPC] TPC. *The TPC-H decision support benchmark*. https://www.tpc.org/tpch/default5.asp. Accessed 2022-10-19 (cit. on pp. 76, 95).

[Tuc+03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. "Exploiting Punctuation Semantics in Continuous Data Streams". In: *IEEE Trans. Knowl. Data Eng.* 15.3 (2003), pp. 555–568. DOI: 10.1109/TKDE.2003.1198390. URL: https://doi.org/10.1109/TKDE.2003.1198390 (cit. on p. 39).

[Tuc+08] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. *Nexmark–a benchmark for queries over data streams (draft)*. Tech. rep. Technical Report. Technical report, OGI School of Science & Engineering at ..., 2008 (cit. on p. 13).

[Vel14] Todd L. Veldhuizen. "Transaction Repair: Full Serializability Without Locks". In: *CoRR* abs/1403.5645 (2014). arXiv: 1403.5645. URL: http://arxiv.org/abs/1403.5645 (cit. on p. 105).

[Wan+17a] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, et al. "The Myria Big Data Management and Analytics System and Cloud Services." In: 2017 (cit. on pp. 105, 111).

[Wan+17b] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. "Graspan: A Single-machine Disk-based Graph System for Interprocedural Static Analyses of Large-scale Systems Code". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. Ed. by Yunji Chen, Olivier Temam, and John Carter. ACM, 2017, pp. 389–404. DOI: 10.1145/3037697.3037744. URL: https://doi.org/10.1145/3037697.3037744 (cit. on pp. 107–109).

[Wei19] Aaron Weiss. "Oxide: The Essence of Rust". In: *CoRR* abs/1903.00982 (2019). URL: http://arxiv.org/abs/1903.00982 (cit. on p. 49).

[Wen07] Makarius Wenzel. "Isabelle/Isar—A generic framework for human-readable proof documents". In: *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*. Ed. by Roman Matuszewski and Anna Zalewska. Vol. 10(23). Studies in Logic, Grammar, and Rhetoric. Uniwersytet w Białymstoku, 2007 (cit. on p. 163).

[Wik] Wikipedia contributors. *Antichain*. https://en.wikipedia.org/wiki/Antichain. Accessed 2022-10-17 (cit. on p. 38).

[Wil+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. "Verdi: a framework for implementing and formally verifying distributed systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 2015, pp. 357–368. DOI: 10.1145/2737924.2737958. URL: https://doi.org/10.1145/2737924.2737958 (cit. on p. 136).

[YSZ17] Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. "Scaling up the performance of more powerful Datalog systems on multicore machines". In: *VLDB J.* 26.2 (2017), pp. 229–248. DOI: 10.1007/s00778-016-0448-z. URL: https://doi.org/10.1007/s00778-016-0448-z (cit. on p. 105).

[Yu+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 1–14. URL: http://dl.acm.org/citation.cfm?id=1855741.1855742 (cit. on pp. 43, 80).

[Zah+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 2–2. eprint: https://scholar.google.comhttps://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf (cit. on pp. 15, 40, 43, 78).

[Zah+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737. URL: http://doi.acm.org/10.1145/2517349.2522737 (cit. on pp. 15, 41, 44, 74, 119).