Distributed Dynamic Partially Stateful Dataflow

by

Jonathan Behrens

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 23, 2018

Certified by . . . . . .
M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . . . .
Malte Schwarzkopf
Postdoc
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Distributed Dynamic Partially Stateful Dataflow

by

Jonathan Behrens

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2018, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis present a distributed implementation of Noria, a new streaming dataflow system that simplifies the infrastructure of read-heavy web applications by unifying the database, caching layer, and parts of application logic in a single system. Noria's *partially-stateful dataflow* allows it to evict and reconstruct state on demand, and avoid prior dataflow systems' restriction to windowed state. Unlike existing dataflow systems, Noria adapts on-line to schema and query changes, and shares state and computation across related queries to eliminate duplicate effort.

Noria's distributed design enables it to leverage the compute power of an entire cluster while providing high availability thanks to its fault tolerant design. On a single machine, Noria already outperforms MySQL by up to 7×, but when running across a cluster of machines, it can scale to tens of millions of reads and millions of writes per second.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Malte Schwarzkopf
Title: Postdoc

Acknowledgments

This thesis would not have been possible without the support and guidance of many people, to whom I am deeply grateful.

From the beginning Noria has been a joint collaboration between a large number of people. When I joined Jon and Malte already had a many clever insights and were well along the way to putting them into effect. I am grateful for their help learning the codebase and getting up to speed in Rust. Throughout the course of development, we've of all benefited greatly from the advice and direction that Frans, Robert and Eddie gave to the project. And of course Noria would not be where it is today without Lara, Martin and the contributions of everyone else who's worked on it.

I would especially like to thank Frans and Malte for their guidance in writing this thesis. Their feedback on how to best present this work has been invaluable.

Lastly I would like to think my friends and family for their dedication and support. Without them this thesis would not have been possible.

Contents

1 Introduction **9**

1.1 Partially Stateful Dataflow 10

1.2 Goals 11

1.3 Challenges 12

1.4 Contributions 13

1.5 Outline 14

2 Related Work **15**

2.1 Dataflow systems 15

2.2 Stream processing systems 16

2.3 Materialized Views 16

3 Partially Stateful Dataflow in Noria **17**

3.1 Target applications and deployment 17

3.2 Programming interface 18

3.3 Dataflow execution 19

3.4 Consistency semantics 21

4 Distribution **23**

4.1 Domains and Vertices 24

4.1.1 Base vertices 25

4.1.2 Ingress vertices . . . 25
4.1.3 Reader vertices . . . 25
4.2 Initialization . . . 25
4.3 Distributed Migrations . . . 26
4.4 Failure Recovery . . . 27
4.4.1 Controller Failure . . . 28
4.4.2 Worker Recovery . . . 28
4.5 Implementation . . . 29

5 Evaluation **31**
5.1 Experimental Setup . . . 31
5.2 Single Machine Performance . . . 31
5.3 Scaling . . . 33
5.4 Recovery . . . 35

6 Transactions **37**
6.1 Challenges . . . 37
6.2 Design . . . 38
6.2.1 Timestamp assignment . . . 38
6.2.2 Dataflow Processing . . . 39
6.3 Alternatives and Future Work . . . 39

7 Conclusion **41**

Chapter 1

Introduction

Web services are frequently built using a cluster of cache servers which store computed queries over a central database that serves as a "source of truth." This design has good properties: the cache servers enable extremely high read load (important since the vast majority of requests to such services are reads), and the central database provides expressive queries including joins and aggregations.

Unfortunately, several complications make application deployment much harder in practice. Whenever any data used to compute any part of a cached query result changes, the result must be evicted or suitably updated across the entire cluster. Special care must also be taken to avoid the "thundering herd" problem where a missing cache entry causes many clients to simultaneously attempt to populate the entry by issuing queries to the central database [35]. Such a thundering herd would otherwise overload the database, leading to poor quality of service for all clients interacting with it.

The Noria datastore provides a principled approach to this problem of efficiently maintaining an up-to-date (but still eventually consistent) cache by integrating database and caching components into a single system. It uses a combination of techniques from the literature on databases and dataflow systems to efficiently maintain cached results for a set of predeclared queries. Central to the design is the idea of *partially stateful dataflow,* which enables Noria to keep only some of its

state in memory and compute the rest on demand. This is crucial since applications may cache results for many queries, and the data may grow over time even if the working set is fixed in size.

The goal of this thesis is to introduce the design and methods to deploy partially stateful dataflow in a *distributed* setting. The rest of this chapter describes partially stateful dataflow in more detail and presents our approach to meeting this goal.

1.1 Partially Stateful Dataflow

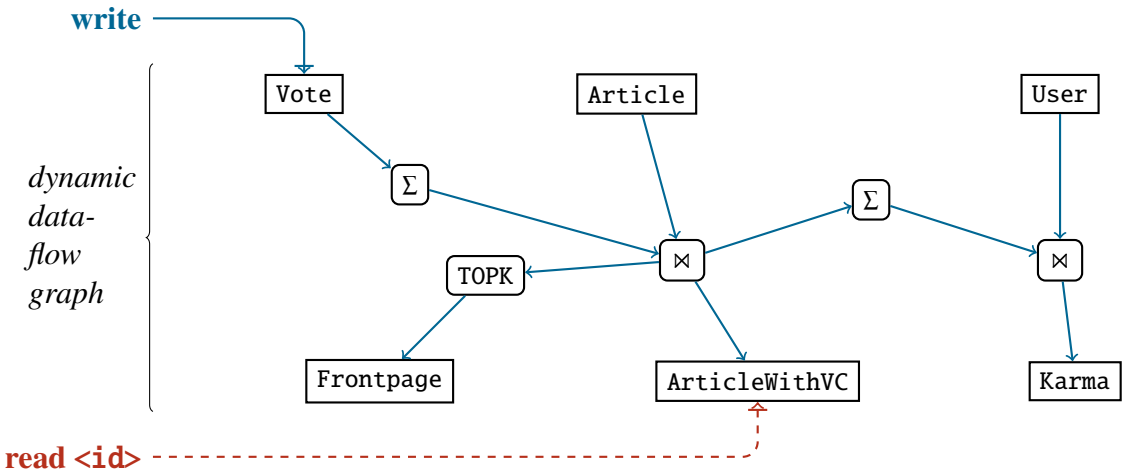


Figure 1-1: An example dataflow graph. Writes enter the system at the top, propagate through the graph, and ultimately update materialized state. Reads directly interact with those output materializations.

A dataflow graph consists of a set of "operators" (*i.e.*, vertices) along with "edges" which define the path that updates "flow" along as they are being processed. Updates define changes that are applied to operator state and in Noria take the form of either row insertions or removals. Once the dataflow graph has been constructed, the processing of individual updates is cheap. Per record computation can be as low as a handful of instructions at each operator, can be easily pipelined, and updates can be batched when moving between operators to amortize network overhead.

Traditionally, operators in dataflow systems are either fully (i) materialized, meaning that they store state reflecting the the complete sequence of update that the operator has seen, (ii) stateless, meaning that they store no state at all, or (iii) windowed, in which case they only reflect the last

k updates for some window k. Partially stateful dataflow introduces another possibility: operators can be partially materialized, conceptually reflecting all prior updates, but in reality only keeping a cache. In normal processing, updates that do not impact any cached state in an operator are discarded without forwarding them onward through the graph. In the event that a client lookup misses in an external materialization's cache, a "backfill" computes the missing entries on demand.

At a high level, a backfill reconstructs a missing cache entry by contacting a nearby ancestor of the operator and directing it to forward along all the records to compute that missing entry. These records then travel along the normal forward processing path interleaved with normal updates until they reach their target. Backfills can be triggered recursively if the nearest ancestor also experiences a cache miss, but must terminate at the base tables which necessarily cannot experience misses.

Noria is especially well-suited for read heavy applications because of the low cost of serving reads in this design. In the common case, reading from an external materialization is just a matter of a hash table lookup; in the worst case, the processing required to perform backfills is still comparable to the query processing in a traditional database, but often much cheaper.

1.2 Goals

In order for Noria to scale to the kind of applications that are currently served by a hybrid database-cache architecture, it must be able to leverage the compute resources of an entire cluster of machines. This thesis describes the design decisions and technical insights involved in making that a reality. In particular, we focus on the following specific objectives:

1. **Scale to Higher Read and Write Throughput.** The objective in making Noria distributed is to enable it to serve a much higher request load than a single machine. Ideally, adding machines should result in linear scaling. This requires particular care to ensure that coordination overheads do not become prohibitive as the cluster size increases.

2. **Survive Failures.** As an application is distributed across more machines, the risk of failure increases dramatically. At the same time, more machines also means a larger total application

state and a higher opportunity cost of downtime. Noria should be able to recover quickly from failures and have the cost of recovery be proportional to the part of the system impacted by the failure.

3. **Design for Supporting Transactions.** Not all applications can tolerate the weak semantics provided by eventual consistency, so Noria should have a way to accommodate them. We introduce a design for single table optimistically concurrent ACID transactions compatible with Noria's overall architecture.

1.3 Challenges

Scalability and efficient recovery are challenging to achieve in a dataflow setting. Some state-of-the-art systems must either restart the computation on failure [26], or take synchronized checkpoints [32]. Existing stream-processing systems typically execute only a single computation, so it is tolerable to restart them on failure, while a Web application backend must continue processing requests for queries unaffected by a failure even while it recovers the state for failed servers.

Partially stateful dataflow also adds its own challenges. Backfills of missing data must be efficient because they can be on the critical path for reads, but may require gathering data from other machines. And even getting the data to the right place isn't enough, backfills also must be compatible with concurrent forward dataflow processing. Otherwise, records could be duplicated or lost, ultimately corrupting application state.

Additionally, Noria depends on being able to alter the dataflow graph while the computation is running in order to adapt to new application queries. This capability is even more important in a distributed setting because application state is larger. However, it is also more difficult to achieve because system state is distributed across many components.

In contrast to many existing streaming dataflow systems that have fixed membership specified at startup time, Noria also needs to support flexible cluster membership so that new machines can be added — both to replaced failed machines and to add capacity for handling more queries.

1.4 Contributions

This thesis makes the following contributions:

1. A design and implementation of partially-stateful distributed dataflow using (i) sequential process-to-completion update handling, and (ii) static graph analysis to establish connectivity (§4). Avoiding concurrent update processing within a single operator decreases operator complexity and reduces synchronization overheads. Meanwhile, establishing connectivity in advance simplifies the data plane.

2. A control plane design and vertex API for dynamic distributed dataflow changes while remaining live for serving unaffected client requests (§4.3). In this design, much of the operator initialization happens in advance in the controller process which has access to global knowledge about the dataflow, which avoids costly global coordination.

3. Techniques for recovering the dataflow from machine failures, implemented by reusing Noria's dynamic dataflow change capabilities, in contrast to the more traditional approach of having completely separate dedicated logic for failure response (§4.4). This method represents a trade-off in terms of recovery time, but avoids the runtime overhead of taking regular checkpoints or replicating dataflow state.

4. An evaluation of our distributed implementation of Noria and a comparison to Differential Dataflow, demonstrating the merits of the design (§5).

5. An extension to the base distributed design to support transactions using fine grained optimistic concurrency control (§6).

The system as currently implemented has several limitations. Dynamic re-sharding and sharded shuffles are currently unsupported, and Noria is currently unable to replicate on-disk state. We hope to address these limitations in future work.

1.5 Outline

This thesis continues with an overview of related work (§2) and then a description of partially stateful dataflow (§3). Next it describes the technical details involved in implementing partially stateful dataflow in a distributed setting (§4) and evaluates the resulting performance (§5). Finally, it describes how to provide efficient transactions in the same setting (§6) and concludes (§7).

Chapter 2

Related Work

System with Noria's performance and adaptability are in demand. For example, the New York Times uses Apache Kafka [5] to achieve some of Noria's flexibility properties [40]. Similar ideas have also been proposed as an extension proposal for Samza [22], but to our knowledge, no prior system achieves the performance and flexibility of Noria.

2.1 Dataflow systems

Dataflow systems excel at data-parallel computing [20, 32], including on streams, though none target Web applications directly. These systems only achieve low-latency incremental updates if they keep full state in memory; Noria and its partially-stateful dataflow are the first to lift this restriction. A few systems can reuse work automatically: *e.g.*, Nectar [15] detects similar subexpressions in DryadLINQ programs, similar to Noria's automated operator reuse, albeit using DryadLINQ-specified merge and rewrite rules. CIEL [33] and Spark [45] can dynamically adapt batch-processing dataflows; Noria does so for long-running streaming computations.

Current dataflow systems use a range of techniques for distribution. Spark and Flink [8] use dynamic task assignment, whereas Naiad [32] and Differential Dataflow [26] places a shard of every operator on each machine. Checkpointing is a popular method for fault tolerance: Naiad uses logical timestamps for snapshotting state, while Flink uses a variant of the Chandy-Lamport

algorithm [9] along with stream alignment to to produce consistent checkpoints without global synchronization. By constrast, Noria's recovery design more closely resembles re-execution based fault tolerance designs like Spark's lineage based recovery. Other systems target short-lived computions and thus lack fault tolerance entirely [26].

2.2 Stream processing systems

Stream processing systems [3, 8, 23, 42, 44] often use dataflow, but have static queries that process only new records. STREAM [6] identifies opportunities for operator reuse, but it only supports windowed state and lacks support for query change. S-Store [29] combines a classic database with a stream processing system using trigger-based view maintenance with the goal of enabling transactional processing, but lacks Noria's partially stateful dataflow.

2.3 Materialized Views

Database materialized views [16, 25] were devised to cache expensive analytical query results; and view maintenance techniques are subject of considerable research [17, 24, 25, 41, 46, 47]. Yet, materialized view support in commercial databases is limited: *e.g.*, SQL Server's indexed views [1] are severely restricted [30, 38], and must be rebuilt to change. DBToaster [2, 34] supports incremental view maintenance under high write loads with compiled recursive delta queries; Noria achieves similar performance and can change queries while live. Pequod [21] and DBProxy [4] support partial materialization in response to client demand, though Pequod is limited to static queries.

Chapter 3

Partially Stateful Dataflow in Noria

Noria is a partially stateful, dynamic, parallel, and distributed dataflow system designed to meet the persistent storage, query processing, and caching needs of typical Web applications.

3.1 Target applications and deployment

Noria targets read-heavy applications that tolerate eventual consistency, but which have some complex query needs. Many Web applications fit this model: they deploy caches to make common-case reads fast and accept the consequent eventual consistency [11, 13, 35, 43]. Noria's current dataflow design primarily targets relational operators, rather than iterative or graph computations [28, 32], and operates on structured records of application data in tabular form [10, 12]. Large blobs (*e.g.*, videos, PDF files) are not directly supported by Noria, but are intended to be stored on external systems [7, 14, 31] and included by reference.

Noria runs on a multicore server and iteracts with clients using RPCs. Noria stores both *base tables* and *derived views*. Roughly, base tables contain the data that a typical database would store persistently, and derived views hold data the application might cache elsewhere. Compared to a conventional stack, Noria base tables might be smaller: Noria derives some data that a conventional application might pre-compute and stores in base tables. Views, by contrast, will likely be larger than a typical cache footprint, because Noria derives more data, including some intermedi-

```
/* base tables */
CREATE TABLE stories
  (id int, author int, title text, url text);
CREATE TABLE votes (user int, story_id int);
CREATE TABLE users (id int, username text);
/* internal view */
CREATE VIEW VoteCount AS /* vote count per article */
  SELECT story_id, COUNT(*) AS vcount
    FROM votes GROUP BY story_id;
/* external view */
CREATE EXT VIEW StoriesWithVC AS /* story details */
  SELECT id, author, title, url, vcount
    FROM stories
    JOIN VoteCount ON VoteCount.story_id = stories.id
   WHERE stories.id = ?;
```

Listing 3.1: Noria program for the Lobsters (`http://lobste.rs`) news aggregator where users vote for stories.

ate results. Noria stores base tables persistently on disk but the resident portion of views are held in server memory. The application's working set in these views should fit in memory for good performance, but Noria can evict infrequently-accessed data or never store it in views at all.

3.2 Programming interface

Applications interact with Noria via an interface that resembles prepared SQL statements. The application registers queries which include parameters to be filled in at execution time (denoted by a question mark). Listing 3.1 shows an example set of queries for a Lobsters-like news aggregator application. This *Noria program* includes base table definitions, *internal* views to be used as shorthand in other expressions, and named *external* views that the application will later query. Internally, Noria instantiates a dataflow to continuously process this program.

To execute a read query, the application supplies Noria with an external view name (*e.g.*, `StoriesWithVC`) and one or more sets of parameter values. Noria then responds with the cached records that match those values. To modify records in base tables, the application issues updates in the form of `INSERT`, `UPDATE`, or `DELETE` statements. Noria applies these to the appropriate base table and then updates dependent views.

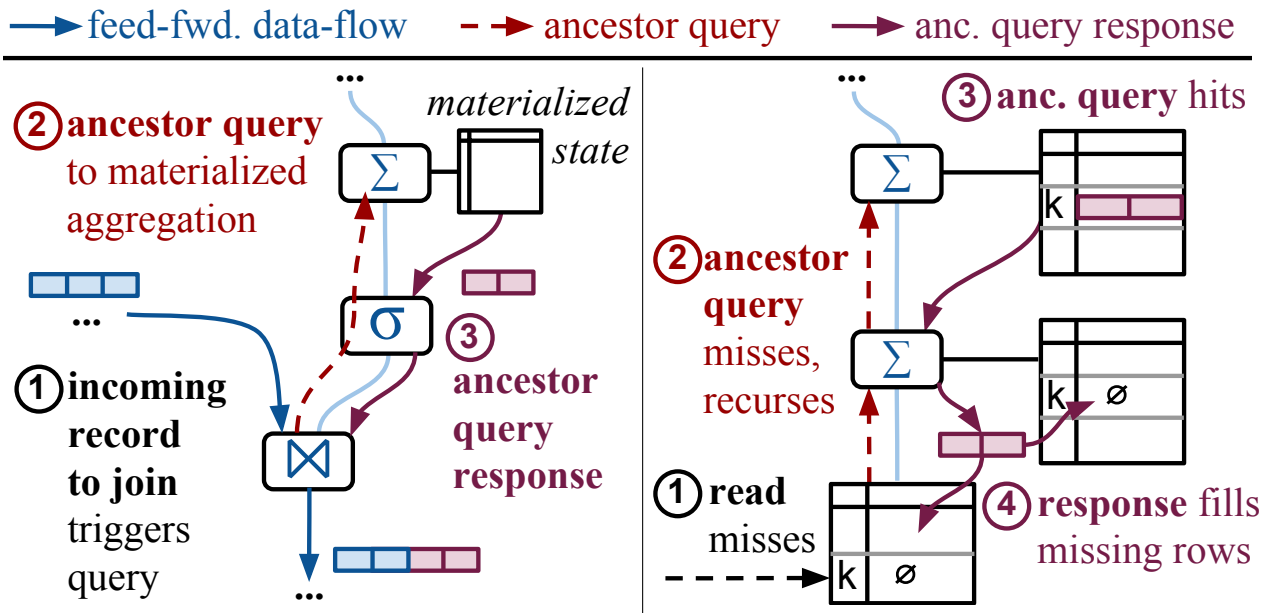
The application may change its Noria program to add new views, modify or remove existing ones, or to adapt base table schemas. Noria expects such changes to be common—*e.g.*, triggered by new SQL queries—and aims to complete them quickly. This contrasts with most previous dataflow systems, which lack support for efficient changes to the computation itself. In addition to its native query interface, Noria provides a backward-compatible SQL interface through a MySQL protocol adapter. The adapter turns ad-hoc queries and prepared statements into writes to base tables, reads from external views, and Noria program changes. It supports much, but not all, SQL sytax.

3.3 Dataflow execution

Noria's dataflow is a directed acyclic graph of relational operators such as aggregations, joins, and projections. Base tables are at the roots of this graph, and external views form the leaves. When a write arrives, Noria applies it to a durable base table and injects it into the dataflow as an *update*. Operators process the update and emit derived updates to their children; eventually updates reach and modify the external views.

Updates are *deltas* [28, 36] that can add to, modify, and remove from downstream state. For example, a count operator emits deltas that indicate how the count for a key has changed; a deletion from a base table generates a "negative" update that revokes derived records; and a join may emit an update that installs new rows in downstream state. Negative updates remove entries when Noria applies them to state, and retain their negative "sign" when combined with other records (*e.g.*, through joins).

Noria supports *stateless* and *stateful* operators. Stateless operators, such as filters and projections, need no context to process updates. Stateful operators, such as count, min/max, and Top-k, maintain state for efficiency (*i.e.*, because re-computing aggregated values is expensive). However, in most situations, stateful operators can maintain *partial* state rather than full state. Partial state supports eviction and can greatly reduce Noria's memory size.



(a) A join uses an ancestor query to retrieve records from upstream dataflow state.

(b) A partially-stateful operator issues a recursing backfill.

Figure 3-1: Operators in the Noria dataflow graph can query into upstream state; this also helps fill missing state in partial operators through backfills.

Stateful operators and joins incur *indexing obligations* on their own and/or their ancestors' state. For example, a count operator that counts votes by user ID has a local indexing obligation on user ID, as efficiently processing a vote update requires looking up the user's current count. Noria tracks obligations through the dataflow and instantiates the corresponding indexes, which, like all state, are partial when possible.

Joins connect multiple dataflow paths; when an update arrives on one path, the join must look up all corresponding records from the others. Most other stream processors solve this problem using windowed state at join inputs, but Noria's views and operator state are not windowed (since applications can request any available data). Instead, Noria joins incur indexing obligations on their *ancestors*, and to process an update, a join performs an *ancestor query* (a request for data from a stateful ancestor). Ancestor query responses flow forward through the dataflow until they reach the querying operator (Figure 3-1a). Though these responses superficially resemble updates, Noria constrains their propagation, and ensures that they only reach the querying operator.

When an application adds a new view, Noria constructs a dataflow representation of it and integrates it with the running dataflow. For example, if Lobsters added a `Karma` view with the total number of votes for each user's stories, Noria might create a dataflow operator that derives from existing views and maintains a sum. Dataflow construction resembles database query planning, but Noria diverges from traditional query planners as it produces a long-term *joint* dataflow across *all* expressions in the Noria program. For instance, if two expressions include the same join, the dataflow contains it only once.

3.4 Consistency semantics

By default, Noria's dataflow operators and the contents of its external views are *eventually consistent*; an update injected by a base table may take some time to propagate, and may appear in different views at different times. Eventual consistency is attractive for performance (*e.g.*, because views need not synchronize to expose updates) and scalability (*e.g.*, because no total ordering of writes is required), and is sufficient for many Web applications [11, 35, 43]. Noria does, however, ensure that all external views eventually expose results that are consistent with applying all the writes received by the system. Making this work correctly requires some care, *e.g.*, to avoid races between updates and ancestor queries. Moreover, since Noria operators with multiple ancestors may see updates in any order, they must commute over their inputs, as standard relational operators do. For applications which cannot tolerate only eventual consistency, This thesis also introduces methods by which partially stateful dataflow systems can support stronger consistency models (§6).

Chapter 4

Distribution

Noria uses a controller-worker model for distribution as shown in Figure 4-1. The controller is selected using a leader election coordinated using Zookeeper and maintains global knowledge about the state of the graph. It stores any persistent state it needs directly inside Zookeeper and also serves as a directory for clients to learn the network addresses for issuing read and write requests. The remaining machines act as workers and accept distinct parts of the graph to handle.

The controller waits for a quorum of workers to join at startup, but additional workers can join later on. A worker joins by reading the the current controller from Zookeeper and then connecting to it directly. Each worker runs a RPC server listening for control messages which can include directives to start new dataflow vertices or modify existing ones. Workers also serve reads from remote materializations, and facilitate other ad-hoc communication between different components in the system.

Noria stores base tables on the local disks of the workers running them, but Noria could instead replicate them using a distributed file system for higher availability. Such a change might slightly degrade write latency but should have no impact on overall throughput.

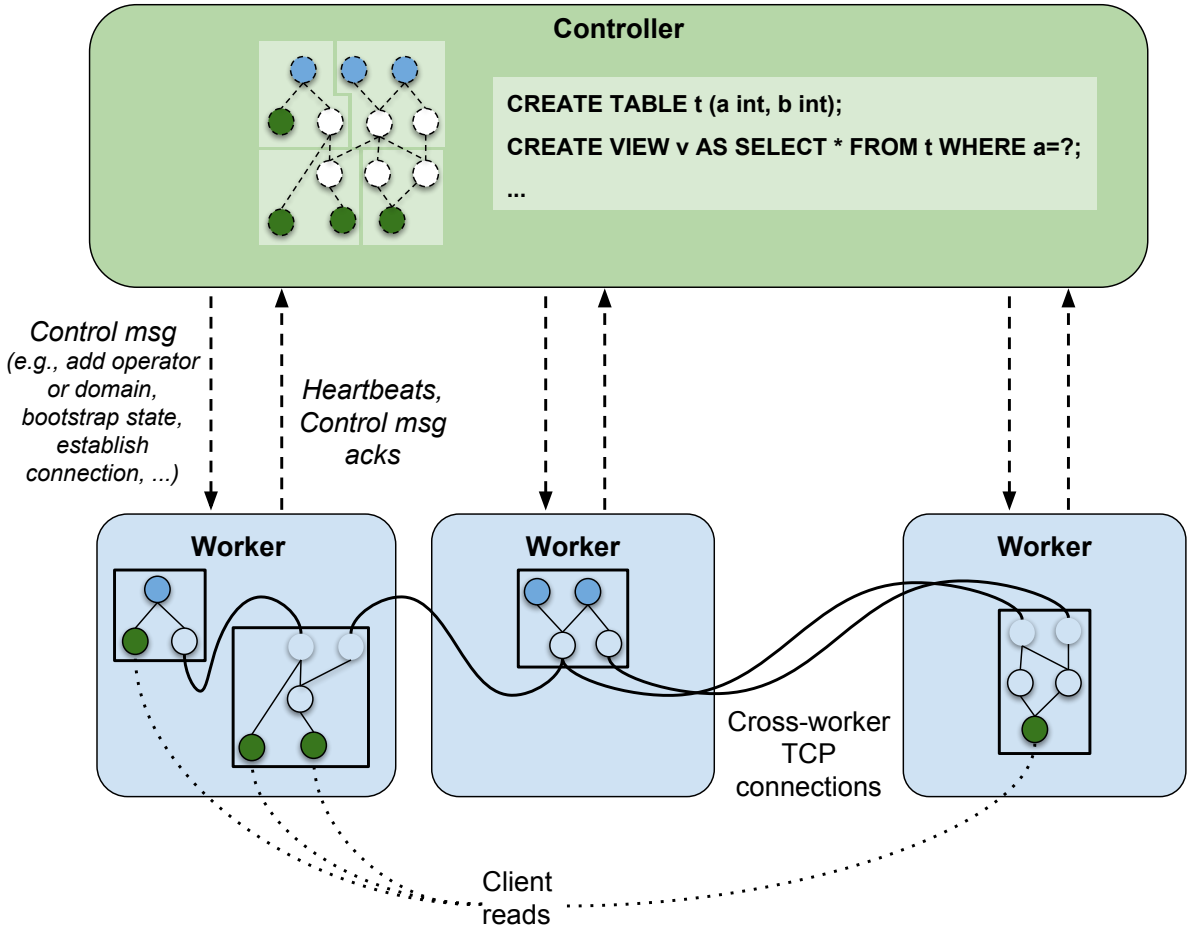


Figure 4-1: Noria's distributed system design. The controller keeps a copy of the current set of queries, as well as the full system dataflow state. The workers each manage a part of the dataflow graph and do process all associated update and read processing.

4.1 Domains and Vertices

Domains are cliques of dataflow vertices. All vertices in a domain are processed by a single thread and each update in a domain in processed to completion before starting the next. In addition to the vertices themselves, domains also store the associated intermediate and output materializations. These design decisions simplify operator logic and avoid synchronization overhead. They also make Noria conducive to a distributed implementation because different domains can be processed entirely in parallel. Another notable property of domains is that the entirety of a domain's state

can be fully reconstructed from only the contents of the base tables. This means that domains do not need to track any additional durable state. In addition to normal dataflow operators, domains contain a couple special kinds of vertices which we describe next.

4.1.1 Base vertices

Base vertices (dark blue in Figure 4-1) are special vertices in the dataflow which represent base tables. They are the first vertices to process any incoming writes, and their role is to persist each write to disk before it propagates out to the rest of the dataflow graph. Ideally, base tables should be written to a distributed (and replicated) file system, to allow recovery on machine failure.

4.1.2 Ingress vertices

Ingress vertices receive updates from other domains in the system and forward them to the other vertices in the domain. They also often store materializations of their ancestors to avoid costly cross-machine lookups that might be required for their children's ancestor queries.

4.1.3 Reader vertices

Noria stores output materializations in reader vertices (shown in green in the figure). Unlike other kinds of vertices whose materializations are designed for efficient single threaded lookups, readers vertices use a special "double buffered" hash table data-structure optimized for concurrent access. This enables the worker to serve high read loads without blocking write processing.

4.2 Initialization

When a new Noria instance starts, the first thing it does is attempt to create a znode with the path `/controller` in the cluster Zookeeper service. If that succeeds, the instance is considered to have won leader election, so it atomically reads the `/state` znode to learn the previous state of the system and updates that znode to reflect that it is now leader (creating the znode if it doesn't already exist).

In the event that the `/controller` znode already exists, the new instance uses its contents to learn the address of the existing controller and connects to it as a follower. Once connected, the follower will send periodic heartbeats so the controller can make sure it hasn't failed.

Once a quorum of followers have connected, the controller checks whether the `/state` znode previously existed, which would indicate that the system was previously running and has just been restarted. If that is the case, then the controller initiates a *distributed migration* to restore the old dataflow graph. Otherwise, the system begins serving client requests immediately.

4.3 Distributed Migrations

Noria uses the migration process to add and remove queries to the dataflow graph. Clients trigger migrations by sending an RPC indicating which queries should be changed to the controller's external HTTP interface. The controller then takes responsibility for doing the high level planning and coordination associated with running the migration.

To start, the controller parses the added SQL queries, converts them into a middle intermediate representation ("MIR") and then runs multi-query optimization over the new and existing queries to join them into a single dataflow graph. Then it breaks the newly added dataflow vertices into cliques ("domains"), which will be divided among the followers. Special ingress and egress dataflow vertices are inserted on both sides of each edge that spans a domain boundary to deal with serialization and communication between threads or across the network.

Next the controller does the actual assignment of domains to workers by sending them each `AssignDomain` messages. If existing domains are receiving new vertices, such vertices are also added now. In either case, the newly added vertices are temporarily kept disconnected from the dataflow until their materializations can be initialized.

Once all vertices have been added, the controller directs each domain to establish TCP connections to its downstream domains. [1] It then coordinates the construction of materializations for any

[1] For simplicity, domains on the same machine actually also communicate over TCP, although they avoid serialization overheads by sending along pointers instead of serialized message contents.

fully materialized vertices. In the simplest case, this is just a matter of feeding the full contents of its parent's materialization through the associated dataflow operator. If the parent vertex is stateless, the closest materialized ancestor forwards the updates along the entire path from the closest fully materialized ancestor instead.

To make this entire process run smoothly, the controller notifies both the source, target, and any intermediate domains along the path of a replay. Then, when the target domain is ready, it triggers the actual replay by sending a packet to the source domain. Once the replay is complete, the target domain notifies the controller that it is ready to proceed.

Finally, after the controller has heard back about all replays, it sends one last message to each impacted domain to say that the migration is complete. In response, these domains flush any buffered messages whose processing was blocked on materialization replay, and then resumes normal dataflow processing.

4.4 Failure Recovery

Distributed Noria must be able to handle a range of failure situations, including losses of single machines, correlated failures among several machines, and network issues that introduces partitions. We assume that the contents of any individual disks are recoverable in these circumstances, although this limitation (and the associated lack of availability) could be relaxed by using a distributed file system.

Noria relies on exactly-once delivery of dataflow messages to provide its eventual consistency guarantee. In other words, Noria's updates are not idempotent on duplicate application. This invariant is hard to provide in the face of failures because in-flight messages can be lost. Persisting every message before sending it would be prohibitively expensive and so failure recovery must account for the fact that surviving materialization may be left in an inconsistent state. For simplicity, the failure recovery process does not attempt to sort out what materializations may be salvageable and instead discards all materializations and restores them from base tables.

4.4.1 Controller Failure

The recovery process for a controller failure proceeds according to the following stages:

1. First, a round of leader election (using Zookeeper) selects one of the machines to be controller and to coordinate the remainder of the recovery process.

2. Upon seeing that a new controller has been selected, all the other surviving machines connect to it as workers.

3. Once a quorum has joined, the controller initiates a migration from an empty dataflow graph to one with all previously active base vertices and materializations.

4. Each base vertex reads back its log files from before the failure, and replays them into the graph.

5. Once this process is complete, the system has been fully recovered and can once again start serving reads and writes.

4.4.2 Worker Recovery

When a worker machine fails, Noria could use the same scheme as for controller failure but this would cause unnecessary disruption. We instead opt for a significantly cheaper method to recover:

1. The controller first detects the failure when it times out waiting for heartbeats.

2. Once a failure is detected, the controller triggers a migration to remove all vertices that are part of queries containing vertices downstream of any vertex on the failed machine.

3. Next, those same queries are added back by a second migration.

4.5 Implementation

Noria is written in 45k lines of Rust, of which approximately 5k are directly related to distributed operation. It uses several libraries including Hyper for HTTP handling and Bincode/Serde for network serialization. We depend on RocksDB [39] for persistent base table storage and Zookeeper bindings for coordination. Workers run thread pools driven by mio and `epoll`.

Chapter 5

Evaluation

We seek to evaluate three questions: how does Noria perform on a single machine compared to MySQL (§5.2)? Can Noria scale linearly in terms of request throughput when running on more machines (§5.3)? And can it recover from failures with recovery work proportional to the lost fraction of graph (§5.4).

5.1 Experimental Setup

The single machine and scaling experiments run on an Amazon EC2 `c5.4xlarge` instance with 16 vCPUs, while the clients run on several `c5.18xlarge` instances to ensure we are never limited by client throughput. We use a "partially open-loop" setup: clients generate load according to a Poisson distribution of interarrival-times, and have a limited number of requests outstanding; additional requests are queued. This avoids backpressure on the clients that can reduce the number of measurements during periods of poor latency [27]. Our test harness measures the "sojourn time" of a request [37], *i.e.*, the time from request generation until it returns from the backend, and the actual achieved request throughput.

5.2 Single Machine Performance

We first evaluate Noria's single machine performance on a realistic Web application workload modeled on production Lobsters traffic statistics [18]. The benchmark emulates authenticated Lobsters

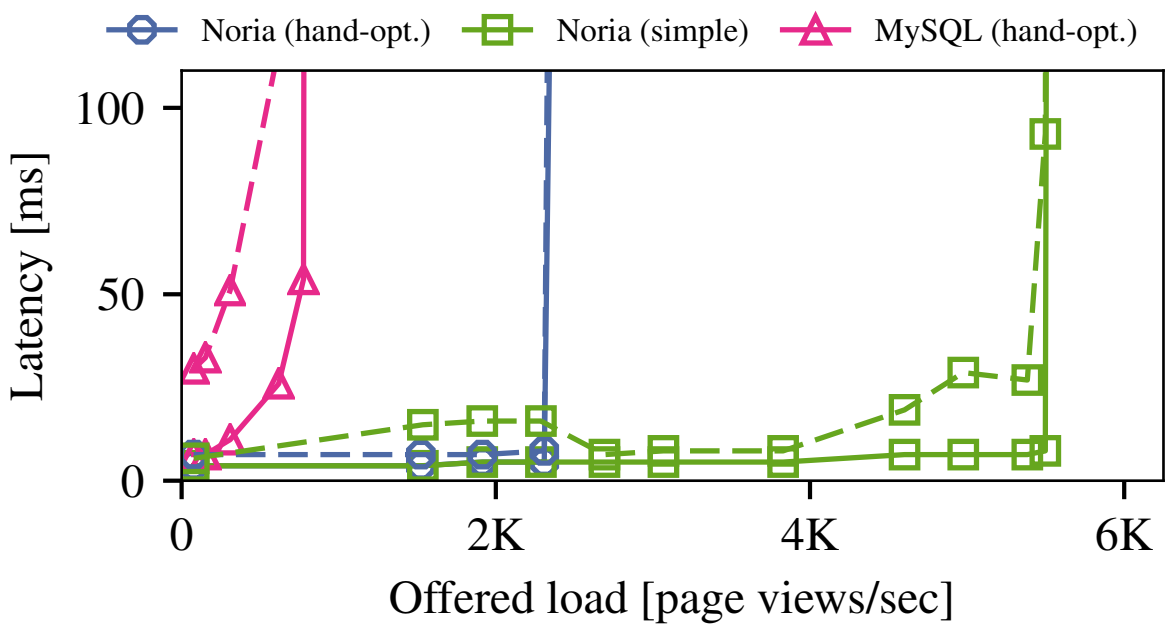


Figure 5-1: Noria scales Lobsters to a 7× higher load than MySQL (3× with the hand-optimized baseline queries). MySQL is limited by read-side computation once it stops scaling, Noria is write-bound. Dashed lines: 95%ile latency.

users visiting different pages according to the access frequencies and popularity distributions in the production workload. Lobsters is a Ruby-on-Rails application, but we emulate only its database interaction, eliminating Rails overhead. We seed the database with 9.2k users, 40k stories and 120k comments, the data size in the real Lobsters deployment, and then run increasing request loads to push the different setups to their scalability limits.

The baseline queries ("hand-opt.") include the Lobsters developers' optimizations, which cache aggregation values and batch reads. We also developed simplified queries that produce the same results using Noria dataflow. MySQL was configured to use a thread pool, to avoid flushing to disk after transactions, and to store the database on a ramdisk to remove overheads unrelated to query execution. With hand-optimized queries, the median page view executes 11 queries; this reduces to eight with simplified queries.

Figure 5-1 shows the results as throughput/latency curves. An ideal system would show as a horizontal line with low latency; in reality, each setup hits a "hockey stick" once it fails to keep up with the offered load. MySQL scales to 700 pages/second, after which it saturates all 16 CPU cores with read-side computation (*e.g.*, for per-page notification counts [19]). Noria running the same queries achieves 3× higher offered load, since its incremental write-side processing avoids redundant re-computation on reads. However, these hand-optimized queries still pre-compute aggregates in the Lobsters application. MySQL requires pre-computation for performance—without it, MySQL supports just 20 pages/sec—but Noria does not. As we hoped, when all aggregate computation is moved into Noria dataflow, performance scales higher still, to 5,400 pages/second (7× MySQL's baseline performance with hand optimized queries). Eliminating application pre-computation reduces overall write load and compacts the dataflow, which lets Noria parallelize that dataflow more effectively. The result achieves good performance and simple, robust queries simultaneously.

5.3 Scaling

We now evaluate Noria's support for distributed operation to see whether Noria can effectively use multiple machines' resources given a scalable workload. We evaluate on a 95%-read, 5%-write workload using a subset of the Lobsters workload; the query schema is the one shown in Listing 3.1. We varied the machine count from 1 to 10 while the graph itself was sharded on the `stories.id` column.

Each machine hosts four shards for writes, leaving 12 cores for reads. For a deployment with N Noria machines, we scale client load to $N \times 5$M requests/second, using the open-loop test harness. This is close to Noria's maximum per-machine load for this benchmark. Load generators select stories uniformly at random, so this workload is perfectly shardable. The ideal result is perfect scalability, with throughput on N servers N times larger than throughput on one. Figure 5-2 shows that Noria supports a near-maximum per-machine load even in a multi-server deployment.

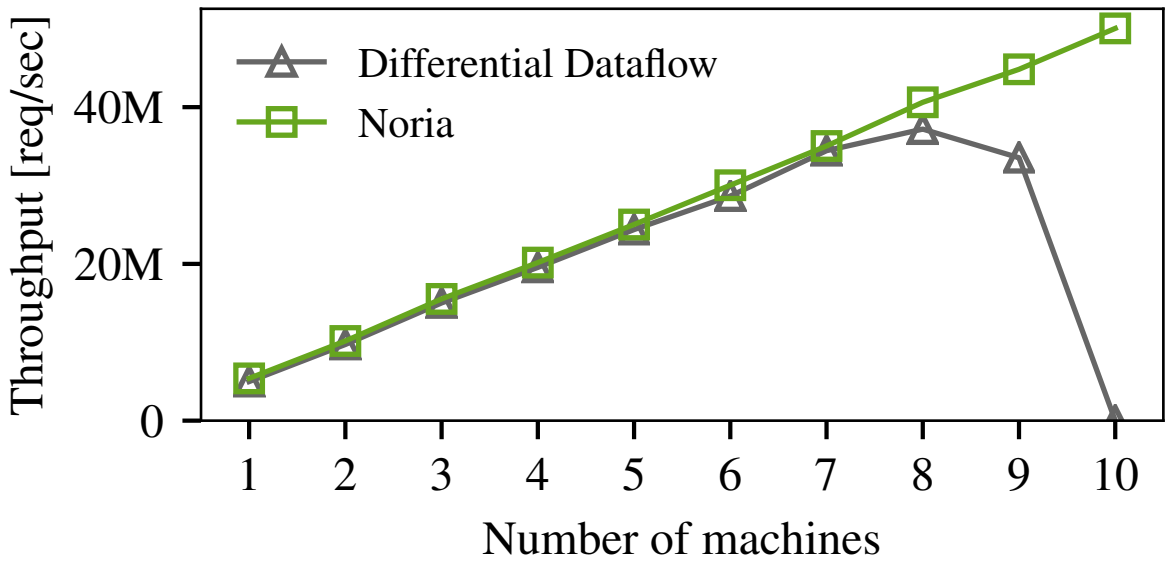


Figure 5-2: On a uniform workload Noria scales well up to 10 machines. Differential dataflow [28] stops scaling at 8 machines due to progress tracking overhead.

We also implemented this benchmark for the Rust implementation of Differential Dataflow (DD) [26], a state-of-the-art system that extends Naiad's ideas [28, 32]. We implemented Listing 3.1 on DD commit `7c1bc9a`. DD lacks a client-facing RPC interface, so we co-locate DD clients with workers; this does not disadvantage DD since load generation is cheap compared to RPC processing. DD uses 12 worker threads and four network threads per machine.

Figure 5-2 shows that Noria is competitive with this advanced scalable dataflow system on this benchmark. Noria's maximum per-machine load, 5M requests/second, is also close to DD's peak load for our latency budget: one DD machine processes that load with 130-ms 95th percentile latency. DD fails to scale beyond seven machines: its progress-tracking protocol, which ensures that writes are exposed atomically, requires workers to coordinate for increasingly smaller batches. Noria avoids this coordination, but only offers eventually-consistent reads.

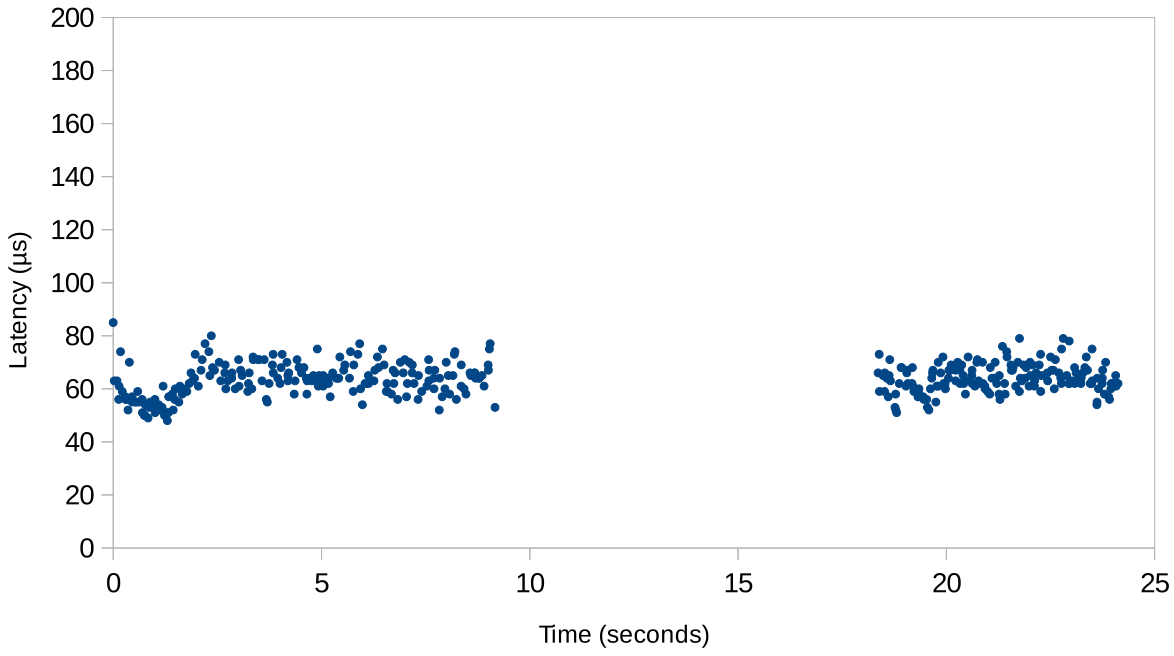


Figure 5-3: Latency serving requests over a loopback interface in the presence of a simulated worker failure. It takes about nine seconds for the fault to be detected and for another instance to take over handling its requests.

5.4 Recovery

Finally, we evaluate how Noria responds to failures. In this experiment, we run the same subset of the Lobsters workload against multiple Noria instances all running with a loopback interface on the same machine (so that the all have access to the base tables). In the middle of the experiment we kill one of the worker processes and observe how long it takes for another instance to take over and resume serving client requests.

The results of this experiment are shown in Figure 5-3. Overall the recovery time was approximately nine seconds from when the fault was triggered to when the system was back to normally handling requests. About half of this time can be accounted to fault detection (the heartbeat timeout was set to four seconds), and the remainder to reconstructing the dataflow graph and replaying the relevant state. While in this particular experiment, Noria assigned all of the dataflow graph to a single domain and thus ran it on a single worker, if the graph consisted of multiple domains then

only the impacted ones would have been recovered, making the cost of recovery proportional to the affected portion of the system.

Chapter 6

Transactions

Noria as described so far provides only eventual consistency guarantees: eventually, if all writes to the system stop, then the contents of views will settle to mutually consistent values. These semantics are sufficient for many applications, particularly because users of Noria do not have to worry about manual materialization and the associated risk that different materializations get perpetually out of sync with each other.

However, there are a class of applications for which eventual consistency is insufficient. For those applications, Noria also provides an operational mode in which is gives stronger guarantees, and exposes a restricted set of ACID transactions. Such transactions may consist of any number of reads followed by a write (of arbitrarily many rows) to a single base table.

6.1 Challenges

To provide transactional semantics, Noria must avoid two sources of inconsistency that could expose partial effects of a write. First, if a dataflow update modifies several records, the changes must appear atomically to transactions. Noria already satisfies this, as domains process each update to completion before handling the next. Second, if a single base table write follows several paths to a multi-ancestor dataflow vertex, Noria must apply the updates atomically and only after they all

arrive. This requirement takes more care to address, and involves logic both during query planning and at runtime.

6.2 Design

In Noria, every base vertex is tagged as either being transactional or non-transactional. Internal dataflow vertices are then considered transactional if all of their ancestors are. Each committed write to a transactional base table is assigned a unique logical timestamp that orders it with respect to every other transactional write. This timestamp is attached to the associated update and alters how it is processed as it propagates through the graph.

Reading from transactional external views produces tokens which concisely encode which writes to the system are reflected in the returned result set. In particular, a token consists of a timestamp along with a list of the base vertices upstream of the external view. It is worth noting that because updates are applied atomically and in order, reads from transactional external views guarantee timeline consistency.

To perform a read-write transaction, the client conducts some number of reads, tracking which tokens it gets back. The set of tokens is then sent back along with with the request write itself. In other words, clients are responsible for explicit specification of the dependencies of each write.

6.2.1 Timestamp assignment

To ensure that they are unique, all timestamps are assigned centrally on a single machine. This machine maintains a *version table* which tracks the timestamps of the last write to each base table in the system. The contents of the version table are used to tell whether a transaction should abort or commit. Specifically, a transaction aborts if any of the base vertices referenced in its tokens have changed since that token was generated.

In addition to the timestamp, transactional updates carry two extra pieces of information. First, they carry the base table of the write that initially triggered the update. This is used during dataflow

processing to determine how many updates each domain will receive with that timestamp (which will be greater than one, if the dataflow forks and rejoins).

Secondly, updates carry the previous timestamp that each domain along the dataflow path will observe prior to processing the update. The information is easily derived from the contents of the version table and global knowledge of the graph. However, by including it in each transactional update, we avoid global communication regarding timestamp tracking: only the parts of the graph impacted by a timestamp need to know that it happened. The remainder of the graph only learns that the timestamp has passed once it gets a later update.

6.2.2 Dataflow Processing

When a domain receives a transactional update, it buffers it until all other updates for that timestamp have been received. Then, once the full set of updates has arrived, the domain processes the whole set together as a batch. Particular care must be taken to send exactly one outgoing message along each link and to ensure that updates to external views only take effect once the entire batch has been processed. If no messages are produced to send on an outgoing link (for instance because they were dropped by a filter), a "null" update must be sent in its place. The merging of outgoing messages is necessary for a subtle but important reason: without it, downstream domains would be unable to predict how many messages they should expect for a given timestamp.

6.3 Alternatives and Future Work

The scheme described so far requires a centralized timestamp assigner. This is unfortunate because the centralized assigner is both a scalability bottleneck and a single point of failure. In future work we hope to investigate alternative options. One promising option is to use vector timestamps; in this design, each base table would have its own entry in the vector, and thus be able to issue timestamps without coordination with other parts of the system.

Chapter 7

Conclusion

This thesis presents and evaluates Noria, a distributed implementation of partially stateful dataflow. Noria is both scalable and reliable. It can support tens of thousands of writes and hundreds of thousands of reads per second across ten machines, and can efficiently recover from both single machine faults as well as whole cluster restarts.

As future work we'd like to extend Noria to support dynamic resharding. Dynamic resharding would enable the system to respond to changing load patterns over time, or to provision an already running instance to handle increased request load by adding more machines. Currently the resources allocated to a query are fixed when the query is created, but if shards of operators could be moved between machines, this limitation would be eliminated.

It would also be desirable for Noria to replicate its in memory state across machines to reduce the downtime caused by failures. Using primary-backup replication, the updates to a domain could be recorded in the memories of several independent machines before being forwarded onward through the graph. This way, if a machines is lost, bringing up a replacement would only be a matter of copying over the state from one of the backups.

Bibliography

[1] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. “Automated Selection of Materialized Views and Indexes in SQL Databases”. In: *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt, Sept. 2000, pages 496–505.

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. “DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views”. In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pages 968–979.

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, et al. “MillWheel: Fault-tolerant Stream Processing at Internet Scale”. In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pages 1033–1044.

[4] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. “DBProxy: a dynamic data cache for web applications”. In: *Proceedings of the 19th International Conference on Data Engineering (ICDE)*. Mar. 2003, pages 821–831.

[5] Apache Software Foundation. *Apache Kafka: a distributed streaming platform*. URL: http://kafka.apache.org/ (visited on 09/14/2017).

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, et al. “STREAM: The Stanford Data Stream Management System”. In: *Data Stream Management: Processing High-Speed Data Streams*. Edited by Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Berlin/Heidelberg, Germany: Springer, 2016, pages 317–336.

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. “Finding a Needle in Haystack: Facebook’s Photo Storage”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pages 1–8.

[8] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. “Apache Flink: Stream and batch processing in a single engine”. In: *IEEE Data Engineering* 38.4 (Dec. 2015).

[9] K. Mani Chandy and Leslie Lamport. “Distributed Snapshots: Determining Global States of Distributed Systems”. In: *ACM Trans. Comput. Syst.* 3.1 (Feb. 1985), pages 63–75.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006.

[11] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al. “PNUTS: Yahoo!’s Hosted Data Serving Platform”. In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pages 1277–1288.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, et al. “Spanner: Google’s Globally Distributed Database”. In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013), 8:1–8:22.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. “Dynamo: Amazon’s Highly Available Key-value Store”. In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, Oct. 2007, pages 205–220.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. “The Google File System”. In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, USA, Oct. 2003, pages 29–43.

[15] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. "Nectar: Automatic Management of Data and Computation in Datacenters". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, 2010, pages 75–88.

[16] Himanshu Gupta and Inderpal Singh Mumick. "Selection of views to materialize in a data warehouse". In: *IEEE Transactions on Knowledge and Data Engineering* 17.1 (Jan. 2005), pages 24–43.

[17] Himanshu Gupta and Inderpal Singh Mumick. "Incremental Maintenance of Aggregate and Outerjoin Expressions". In: *Information Systems* 31.6 (Sept. 2006), pages 435–464.

[18] Peter Bhat Harkins. *Lobste.rs access pattern statistics for research purposes*. Mar. 2018. URL: https://lobste.rs/s/cqnzl5/lobste_rs_access_pattern_statistics_for#c_hj0r1b (visited on 03/12/2018).

[19] Peter Bhat Harkins. *`replying_comments` view in Lobsters*. Feb. 2018. URL: https://github.com/lobsters/lobsters/blob/640f2cdca10cc737aa627dbdf0bbe398b81b497f/db/views/replying_comments_v06.sql (visited on 04/20/2018).

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pages 59–72.

[21] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. "Easy Freshness with Pequod Cache Joins". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, Washington, USA, Apr. 2014, pages 415–428.

[22] Martin Kleppmann. *Turning the database inside-out with Apache Samza*. Mar. 2015. URL: https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html (visited on 05/09/2016).

[23] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia, May 2015, pages 239–250.

[24] Per-Åke Larson and Jingren Zhou. "Efficient Maintenance of Materialized Outer-Join Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Apr. 2007, pages 56–65.

[25] Ki Yong Lee and Myoung Ho Kim. "Optimizing the Incremental Maintenance of Multiple Join Views". In: *Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (DOLAP)*. Bremen, Germany, Nov. 2005, pages 107–113.

[26] Frank McSherry. *Differential Dataflow in Rust*. URL: https://crates.io/crates/differential-dataflow (visited on 01/15/2017).

[27] Frank McSherry. *Throughput and Latency in Differential Dataflow: open-loop measurements*. Aug. 2017. URL: https://github.com/frankmcsherry/blog/blob/master/posts/2017-07-24.md#addendum-open-loop-measurements-2017-08-14 (visited on 04/13/2018).

[28] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. "Differential dataflow". In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.

[29] John Meehan, Nesime Tatbul, Stan Zdonik, et al. "S-Store: Streaming Meets Transaction Processing". In: *Proceedings of the VLDB Endowment* 8.13 (Sept. 2015), pages 2134–2145.

[30] Microsoft, Inc. *Create Indexed Views – Additional Requirements*. SQL Server Documentation. URL: https://docs.microsoft.com/en-us/sql/relational-databases/views/create-indexed-views#additional-requirements (visited on 04/16/2017).

[31] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, et al. "f4: Facebook's Warm BLOB Storage System". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pages 383–398.

[32] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of SOSP*. Nov. 2013, pages 439–455.

[33] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pages 113–126.

[34] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pages 511–526.

[35] Rajesh Nishtala, Hans Fugal, Steven Grimm, et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pages 385–398.

[36] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. "How to Roll a Join: Asynchronous Incremental View Maintenance". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. Dallas, Texas, USA, 2000, pages 129–140.

[37] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, 2006, pages 239–252.

[38] Jes Schultz Borland. *What You Can (and Can't) Do With Indexed Views*. Brent Ozar Unlimited Blog. URL: https://www.brentozar.com/archive/2013/11/what-you-can-and-cant-do-with-indexed-views/ (visited on 04/16/2017).

[39] Facebook Open Source. *A persistent key-value store for fast storage environments*. 2018. URL: http://rocksdb.org/ (visited on 04/20/2018).

[40] Boerge Svingen. *Publishing with Apache Kafka at The New York Times*. Confluent, Inc. blog. Sept. 2017. URL: https://www.confluent.io/blog/publishing-apache-kafka-new-york-times/ (visited on 09/14/2017).

[41] Frank W. Tompa and Joseph A. Blakeley. "Maintaining Materialized Views Without Accessing Base Data". In: *Information Systems* 13.4 (Oct. 1988), pages 393–406.

[42] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, et al. "Storm@Twitter". In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. Snowbird, Utah, USA, June 2014, pages 147–156.

[43] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (Jan. 2009), pages 40–44.

[44] 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 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 423–438.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. "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 (NSDI). San Jose, California, USA, Apr. 2012, pages 15–28.

[46] Jingren Zhou, Per-Åke Larson, and Hicham G. Elmongui. "Lazy Maintenance of Materialized Views". In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. Vienna, Austria, Sept. 2007, pages 231–242.

[47] Yue Zhuge, Héctor García-Molina, Joachim Hammer, and Jennifer Widom. "View Maintenance in a Warehousing Environment". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. San Jose, California, USA, May 1995, pages 316–327.