Read-Your-Writes Consistency in Streaming Dataflow Systems

by
Ishan Sharma
B.Tech., SRM Institute of Science and Technology, 2020

A thesis submitted in partial fulfillment of the
requirements for the Degree of Sc.M.
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2022

© Copyright 2022 by Ishan Sharma

Acknowledgments

I could never thank Malte Schwarzkopf, Kinan Dak Albab, Leonhard Spiegelberg, Theophilus Benson and Franco Solleza enough for helping me become a better thinker. I am certainly not as great as they are, but I have come a long way since I started my Master's. I am also indebted to my friends and family for supporting me in more ways than imaginable over these last two years.

Contents

1 Introduction **1**

1.1 Pelton . . . 1

1.2 Motivation . . . 1

1.3 Summary . . . 2

1.3.1 Other Contributions . . . 2

2 Background and Related Work **3**

2.1 Dataflow systems . . . 3

2.2 Consistency Models . . . 3

2.3 Related Work . . . 4

3 Parallel Dataflow in Pelton **6**

3.1 Data Parallelism . . . 6

3.2 Partitioning and Exchange Operators . . . 6

3.2.1 Need for partitioning . . . 6

3.2.2 Graph traversal . . . 7

3.3 Workers and communication . . . 8

4 RYW via Futures and Promises **9**

4.1 Core Mechanism . . . 9

4.2 Futures and Promises in Pelton . . . 10

4.3 Write Processing with Futures and Promises . . . 10

4.4 Limitations . . . 12

5 Evaluation **13**

5.1 Write processing . . . 13

5.2 Latency . . . 15

6 Conclusion **17**

List of Figures

3.1 Prepared statement to illustrate graph traversal. 7
3.2 Traversal algorithm to insert exchange operators. 7

4.1 Futures and promises in Pelton. 10
4.2 Write processing in parallel dataflow with futures and promises. 11

5.1 Write processing overheads associated with Pelton's RYW consistency mechanism. . 14
5.2 Latency overheads associated with Pelton's RYW consistency mechanism. 15

Chapter 1

Introduction

1.1 Pelton

In traditional storage systems, users' data is scattered across tables and databases. Hence, complying with privacy rights (such as GDPR access and deletion rights) ends up requiring substantial manual effort from the developer's end which if not done correctly can end up violating compliance and risking steep fines. Pelton addresses this by internally sharding data by user, wherein each user's data is present in it's own micro-database (µDB). Hence, a GDPR access/deletion request by a user simply involves fetching/deleting a particular micro-database. Pelton makes queries over these databases feasible and efficient (in the presence of read heavy workloads) via materialized views that are updated by an incremental streaming dataflow computations.

1.2 Motivation

Pelton was initially eventually consistent, i.e., there were absolutely no guarantees as to when a write in a µDB would be reflected in a materialized view. Large scale web applications are typically fine with eventual consistency as this is seen as an acceptable trade-off in order to obtain high availability [6]. Stronger forms of consistency are desirable, and it turns out, necessary for Pelton. The Lobsters application [11], which we use as a primary benchmark for Pelton, conceptually works fine with an eventually consistency backend [7]. However, in case of Pelton, it did not and caused Pelton to crash.

The reason for this crash is that Pelton, in addition to using the dataflow infrastructure to serve applications, also uses it to construct and maintain indices. These indices, among other things are used to perform transitive sharding, which is the process of transitively traversing the foreign keys specified by the database schemas in order to identify a target µDB for a given `INSERT`. For instance, if Pelton cannot identify the target µDB from a given `INSERT`, it uses appropriate information from the `INSERT` query to check the dataflow index (recursively if there are multiple indices whose records

are "linked" together) to deduce the target µDB. In case of Lobsters [11], its client performed two consecutive `INSERT`s such that the first `INSERT` ended up updating the secondary index and the second `INSERT` made use of the updated index to perform transitive sharding, i.e., it expected the previous insert to be reflected in that index. When Pelton tried to execute the second `INSERT` and did not find the expected record in the index, it crashed.

Hence, although the Lobsters application does not require anything stronger than eventual consistency, because of they way Pelton is internally using the dataflow infrastructure, it requires a bare minimum of Read-Your-Writes (RYW) to function. Although this is crucial for Pelton to function, from a broader perspective, RYW is a desirable property for a system to have as it improves both the developer and end user experience [4, 17].

In order for Pelton to offer RYW consistency, the bottom half consisting of the µDB storage and the top half consisting of the streaming dataflow must both offer RYW consistency. Since Pelton uses RocksDB [16] for µDB storage, there is no work to be done over there, but the dataflow on the other hand is eventually-consistent and requires additional support from the infrastructure to provide RYW consistency.

1.3 Summary

To that end the main contribution of this thesis is a lightweight consistency mechanism based on futures and promises to provide RYW consistency in an incrementally updated computational streaming dataflow.

The rest of the thesis is organised as follows. Chapter 2 describes the background and related work. Chapter 3 describes the design of the parallel dataflow in Pelton, which forms the basis of what needs to be done in order to provide RYW consistency in a parallel dataflow. Chapter 4 describes the design of futures and promises and their use to provide RYW consistency in Pelton. Finally, Chapter 5 presents a comprehensive evaluation of the system.

1.3.1 Other Contributions

The work done as part of this thesis is associated with the larger Pelton project that I have been involved with since its inception in Fall 2020. My other contributions to the Pelton project include, building the single threaded dataflow, where I built operators from scratch and optimized them for batch processing, integrating Apache Calcite [3] for query planning into our system, building the parallel dataflow engine for multi-threaded processing, adding support for multi-threading and query re-writing in our MySQL proxy, engineering a write processing dataflow microbenchmark, incremental engineering work for the Lobsters application [11] benchmark and a Memcached based caching performance benchmark, and performance debugging associated with the aforementioned benchmarks.

Our work is currently under submission at NSDI '23 [1].

Chapter 2

Background and Related Work

2.1 Dataflow systems

The term *dataflow system* is used to refer to a broad variety of systems which share the core paradigm of data "flowing" from one section of the system to another. Streaming systems that incrementally maintain state [2, 7, 13, 14, 18] are a key example. However, batch processing systems like MapReduce and Spark also share this paradigm [12]. The primary purpose of both the aforementioned classes of systems is to compute over data as it flows through the system. There also exist systems that do not compute or transform data but share the same dataflow paradigm. Facebook's TAO infrastructure is one such example, where the system is designed to serve as an asynchronous caching pipeline [4].

In the context of Pelton, the term "dataflow system" is used to refer to an incremental streaming system which consists of a sequence of operators that correspond to a query plan of a SQL query. These operators could be stateless (filter, project, input, ..etc.) or stateful (equijoin, aggregate, materialized view). Reads in such a system are fast as they are served directly from materialized views present in memory but this comes at a higher write processing cost.

2.2 Consistency Models

There are a myriad of consistency models applicable to dataflow systems. However, researchers and practitioners usually care about three consistency models:

1. **Eventual Consistency:** This is the weakest form of consistency guarantee that a system can provide without sacrificing correctness. Eventual consistency entails that if no new updates are made to the system, the system will eventually reach a consistent state. When it comes to large scale applications eventual consistency is viewed as an acceptable trade-off in order to provide high availability [4, 6, 7]. One of the crucial drawbacks of eventual consistency is that it is a lot harder for an application developer to reason about the system and that users

may observe incorrect or inconsistent outputs. But providing any stronger guarantees requires more infrastructure support with an associated performance penalty.

2. **Read Your Writes (RYW):** RYW is a stronger form of consistency guarantee where from the perspective of a client, any read will always reflect *all* previous writes performed by the same client. Upon extending the notion of a client to a user-centric session, RYW is a very desirable property to have in a system where an end user interacts with the system via multiple devices. Facebook's motivation to support RYW in TAO is exactly based on this, with an additional and equally important intent to improve developer experience [17].

3. **Strong Consistency:** This is the strongest form of consistency a system can offer. If a system is strongly consistent, it can be viewed as if it operates as a monolith. Strong consistency entails a bare minimum of sequential consistency (serializability in the database realm) and potentially linearizability (a special case of strict serializability in the database realm) [15].

2.3 Related Work

Property	Timely Dataflow [14]	FlightTracker [17]	Pelton
Consistency	Strong	RYW	RYW
Core mechanism	Timestamps	Write-set tracking	Futures and Promises
Client application involved?	Yes	No	No
Decoupled consistency infrastructure	No	Yes	No
Supported in a distributed setting	Yes	Yes	No
Staleness resolution mechanism	Not applicable	Consistency miss	Not applicable
Write synchronicity	Asynchronous	Asynchronous	Synchronous

Table 2.1: Comparison of approaches that provide stronger consistency semantics.

Table 2.1 provides a detailed comparison of various approaches that are used to provide stronger consistency semantics alongside Pelton. The consistency mechanisms in Timely Dataflow [14] and Pelton are baked into their respective systems, whereas FlightTracker [17] is operated as a service and is decoupled from the underlying infrastructure at Facebook.

FlightTracker's RYW consistency mechanism is based on write set tracking where the system issues and maintains a *Ticket* for each client that contains the most recent writes to the versioned social graph. This ticket is included with every read issued to the system, and the system ensures that the read will include the writes specified in the ticket. But it is important to note that TAO [4] is not a computational dataflow system, its primary purpose is to serve as an asynchronous caching pipeline. A write set tracking mechanism in a computational dataflow setting would require non-trivial amount of support from the infrastructure. For instance, a *reverse index* [17] would potentially have to be maintained for each and every flow installed in the system. It is also not trivial to uniquely identify records since columns may be projected out causing the primary key to

get dropped or the record hash to get changed. Aggregations complicate things further as write sets would have to be merged.

Timely Dataflow is based on a timestamp based progress tracking mechanism and offers *strong consistency*. In case of Timely Dataflow, the client is required to announce when it sends data with a new timestamp so that workers can infer that it is done sending data with timestamps lower than the announced timestamp, and make progress on computations. Once crucial advantage of Pelton over Timely Dataflow is that Pelton's consistency mechanism does not impose any such requirements on the client.

Chapter 3

Parallel Dataflow in Pelton

3.1 Data Parallelism

There primarily exist two approaches for building parallel dataflow systems: *pipeline parallelism* and *data parallelism*. In pipeline parallelism, some or all of the operators are executed on different threads [7, 9]. Hence, operators can process different sets of records concurrently.

Pelton along with some other systems [13, 14], on the other hand, make use of data parallelism. In this approach, identical copies of the entire dataflow graph are executed on different threads and data is partitioned across the different threads via a pre-determined partitioning strategy. This approach requires a "shuffle" or "exchange" operation before stateful operators when the partitioning key changes (described in section 3.2).

Pelton's dataflow engine performs the following operations on a given dataflow graph:

1. Clone given input graph into multiple partitions.
2. Determine where exchange operators are required and insert them accordingly.
3. Set up communication channels between partitions.
4. Allocate partitions to worker threads for execution.

3.2 Partitioning and Exchange Operators

3.2.1 Need for partitioning

Partitioning of records is the process of distributing records across the different partitions of the parallel dataflow. Partitioning is necessary because we need to maintain the invariant that all records having the same key are processed by exactly one partition. Pelton makes use of hash partitioning and partitions a record based on a *partitioning key*. A partitioning key is simply a vector of column identifiers.

```
SELECT input1.Col0, COUNT(input1.Col1)
FROM input1
JOIN input2 ON input1.Col0 = input2.Col1
GROUP BY input1.Col1
HAVING input1.Col0 = ?;
```

Figure 3.1: Prepared statement to illustrate graph traversal.

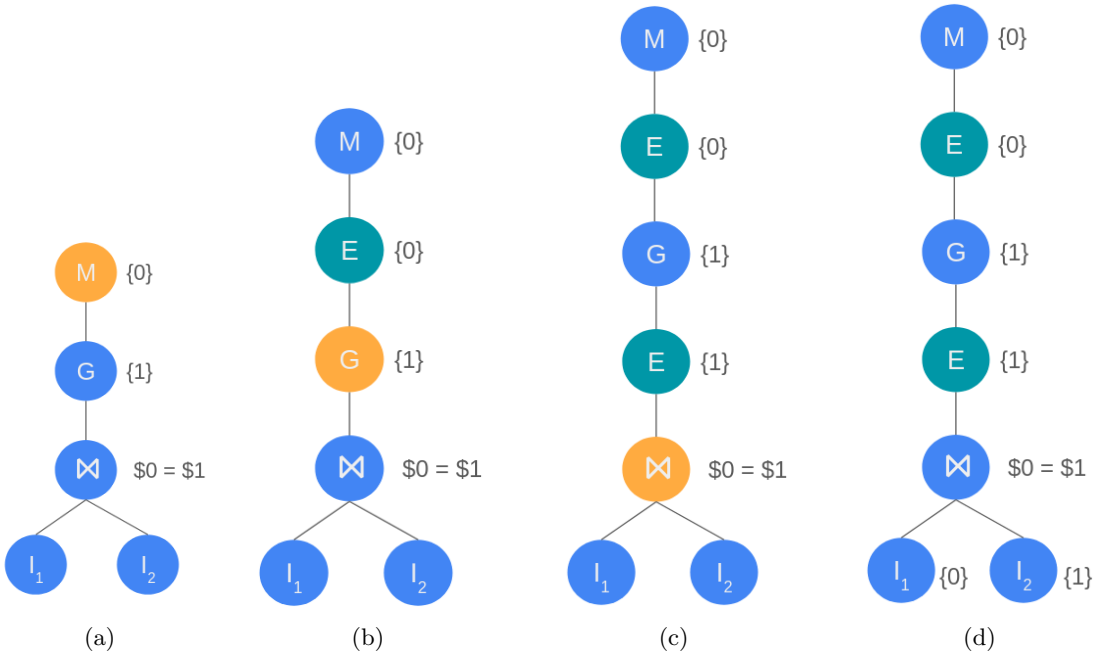


Figure 3.2: Graph traversal for the dataflow of Figure 3.1. The yellow node represents the current operator under consideration, green nodes represent exchange operators and the value inside {} represents the partitioning key. The traversal progresses from (a) to (d).

Exchange Operator: In Pelton, partitioning occurs at two places, once before records are fed into the input operators of the different shards and the second at an exchange operator. The exchange operator's job is to redistribute incoming records and send them to appropriate shards based on an *output key* which is supplied during operator creation. An exchange operator is required before a stateful operator if the input partitioning key of the stateful operator does not match its partitioning requirement.

3.2.2 Graph traversal

When a new flow is installed in the system, as part of the planning phase, the dataflow engine performs a graph traversal on the given dataflow graph in order to figure out where exchange operators are required, and it inserts them accordingly. The traversal algorithm performs a depth first traversal starting at the top on the materialized view and ending at the bottom on the input operator(s). It starts with the partitioning requirements of the materialized view and tries to push

the partitioning key as it traverses downwards. If the operator downstream has different partitioning requirements than the operator upstream, then the traversal inserts an exchange operator with the *output key* of the exchange specified as required by the operator upstream. Consider the prepared statement in Figure 3.1 whose corresponding dataflow graph, very similar to it's query execution plan, is shown in Figure 3.2a. The stateful operators require their input records to be partitioned on the key specified in Figure 3.2a. These keys are a direct consequence of the corresponding prepared statement. The materialized view is indexed on the zeroth column, the aggregate operator aggregates on the first column and the equijoin operator performs a join on the zeroth column of the left input and the first column on the right input. The input records to each of these operators must be partitioned on the aforementioned keys.

The traversal starts from the materialized view in Figure 3.2a. Since the downstream aggregate operator's partitioning key is $\{1\}$, which is not the same as the materialized view's partitioning key, an exchange operator is inserted in Figure 3.2b with $\{0\}$ as it's *output key*. Similarly, another exchange operator is inserted in Figure 3.2c. Finally, since input operators are stateless, the partitioning requirements of the equijoin can be pushed downstream to the input operators resulting in Figure 3.2d. An exchange is not required at the input operators because the dataflow engine partitions records before dispatching them to input operators for processing.

3.3 Workers and communication

Worker threads are responsible for carrying out processing of messages of a particular particular partition of a target dataflow. A message simply contains the target flow name and a vector of records. If the number of available hardware threads are N, when deploying Pelton we configure the number of worker threads to be N and partition each dataflow installed in the system N times (i.e. one partition per worker thread). This setup extracts the most concurrency out of the available hardware resources.

In-memory queues serve as communication channels and are used to facilitate communication between the dataflow engine and worker threads, and the exchange operators with other worker threads. The engine reserves a dedicated channel for each physical input operator and each exchange-worker pair. This setup is much better than an alternative of one channel per worker thread, where the dataflow engine and all the exchange operators end up contending on a single channel. However, in order for a worker to be able to read from multiple channels, the channel has to be non-blocking. In order to accomplish this, a worker wakes up and reads from all the channels whenever a message is enqueued on any of its channels.

Chapter 4

RYW via Futures and Promises

4.1 Core Mechanism

In order to add support for providing RYW consistency in an eventually consistent dataflow system, the key component is to come up with a detection mechanism that would allow the system to detect when a write has been fully applied, and as a consequence, the system could ensure that the write is included on subsequent reads issued by the same client. In case of Pelton, a detection mechanism is required in order to establish as to when a batch of records, fed into the input operator, has reached the appropriate set of materialized views, as a write performed by a client could end up updating many dataflows. Furthermore, this detection mechanism should also extend to a parallel setting (section 3.2). Since a batch of records supplied by the client could be split and sent to multiple partitions, the mechanism should be able to tell when *all* the appropriate partitioned materialized views have been updated. There are several ways of designing such a mechanism, each with their advantages and drawbacks (Chapter 2). The consistency mechanism is based on futures and promises.

Futures and promises have their roots in asynchronous programming. They have a rich history and have subtle variations as to what they mean depending on the programming language they are used in [10]. However, generally speaking, a future is a placeholder for a value that will become available at a later point in time. A future has an associated promise which is responsible for setting the aforementioned value. The fact that futures and promises can be handled by different threads is what gives rise to asynchronicity. After creating a future, a thread can go on with its normal execution while another thread performs necessary computation in order to set the promise and hence *resolve* the future. The thread that owns the future can receive the value either by blocking on the future or by polling it.

```
class Future{
    public:
        Promise GetPromise();
        void Wait();
    private:
        std::atomic<uint16_t> counter_;
        std::binary_semaphore semaphore_;
}
class Promise{
    public:
        void Set();
        Promise Derive();
    private:
        Future *future_;
}
```

Figure 4.1: Futures and promises in Pelton.

4.2 Futures and Promises in Pelton

Pelton currently operates on a single machine and is not distributed. As a consequence, the implementations of futures and promises that we make use of, do not, and are not required to work in a distributed setting. I initially started out with the futures and promises provided by the C++ standard library [5]. However, their use caused an unavoidable deadlock in Pleton's parallel dataflow where two worker threads would end up waiting for each other to set their promises. As a consequence, I implemented custom futures and promises described in Figure 4.1. A *Future* consists of an atomic counter and a binary semaphore, and it exposes *GetPromise()* and *Wait()* APIs. The atomic counter is used to keep track of pending promises and the binary semaphore is used to implement the blocking *Wait()* API. A *Promise* consists of a pointer to the future that it is associated with and it exposes *Set()* and *Derive()* APIs. The *Set()* API decrements the atomic counter of the future that it is associated with, and the *Derive()* API increments the atomic counter and returns a new derived promise. These implementations are highly specific to Pelton's dataflow and are not meant to be generic.

4.3 Write Processing with Futures and Promises

When the dataflow engine, being executed by a thread handling the client's request, receives a batch of records for processing, it looks up the partitioning key for the target input table and then partitions the given input batch based on the partitioning key. There could be multiple dataflows connected to the given input table. Since these flows operate independently, each of these flows will have its own input operator for that particular table. Hence, the dataflow engine creates a copy of records for each flow. In addition to performing these usual operations, as part of the consistency mechanism, the engine creates one future per flow. Then via the *GetPromise()* API it attaches the corresponding promise to their respective batches (one batch per flow) and dispatches them for processing to the workers over their dedicated input channels (Figure 4.2a). Finally, the dataflow

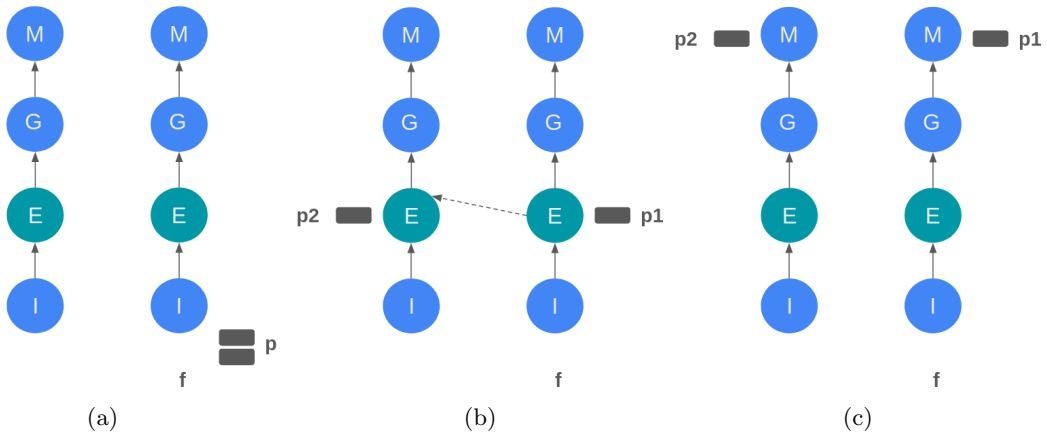


Figure 4.2: Dataflow processing with futures and promises. The engine attaches a promise p to a batch of two records in (a), exchange operator partitions the batch and creates derived promises *p1* and *p2* in (b), and materialized views set promises in (c). A future is resolved if and only if *all* derived promises are set.

engine causes the client thread to block by invoking the *Wait()* APIs of all the previously generated futures. Meanwhile, Pelton's worker threads consume and process these batches in the background.

Since an exchange operator could potentially end up repartitioning a given batch, it internally ends up creating derived promises, one for each partitioned batch, and dispatches them to other partitions for processing (Figure 4.2b). This process of deriving promises occurs recursively for each exchange operator in the dataflow. As and when promises get derived, the atomic counter in the corresponding future gets incremented by one.

When a promise reaches a materialized view (Figure 4.2c), the materialized view sets the promise which causes the counter in the associated future to get decremented by one. If the counter reaches zero then the future is considered to be *resolved* and the thread calling *Set()* on the promise ends up signalling the semaphore. As a consequence, the thread that was blocked on the future wakes up and can continue normal execution. Hence, when the client thread resumes execution, it can be assured that all of its writes have been reflected in the relevant set of materialized views.

Pipelined Processing: If a client thread is blocked on a future, other clients can still dispatch write and read requests to Pelton, causing the promises (included in batch messages) to get pipelined in the channels. This pipelining is however different from promise pipelining [10], which refers to pipelining of promises generated by a single client.

Implementation: Pelton's custom futures and promises library consists of 103 LOC. Although Pelton's dataflow infrastructure consists of 5.3K LOC, as a consequence of the library's specificity to a dataflow setting, only 230 lines of existing code had to be modified in order to add support for RYW consistency via futures and promises.

4.4 Limitations

Pelton's RYW consistency mechanism has two limitations:

1. **Lack of support in a distributed setting:** For a distributed setting, more infrastructure support would be required. Since a shared memory counter would no longer work, a mechanism consisting of child futures would be required where promises would notify their futures via RPCs and upon resolution, child futures would similarly issue RPCs to their parent futures causing this chain to terminate at the client. Child futures would help in spreading out RPCs across different nodes so that there are no inherent bottlenecks. Promises getting dropped as a consequence of failures would also have to be handled. These issues are addressable via tractable engineering effort and are out of scope from the perspective of this thesis.

2. **Limited throughput for a single client:** Depending on what purpose this dataflow is being used for, the nature of the load, and the batch size being used, a single client could observe lower throughput when compared to using eventually consistent Pelton. However, per-client throughput is typically secondary in the context of large scale web applications (elaborated in Chapter 5).

Chapter 5

Evaluation

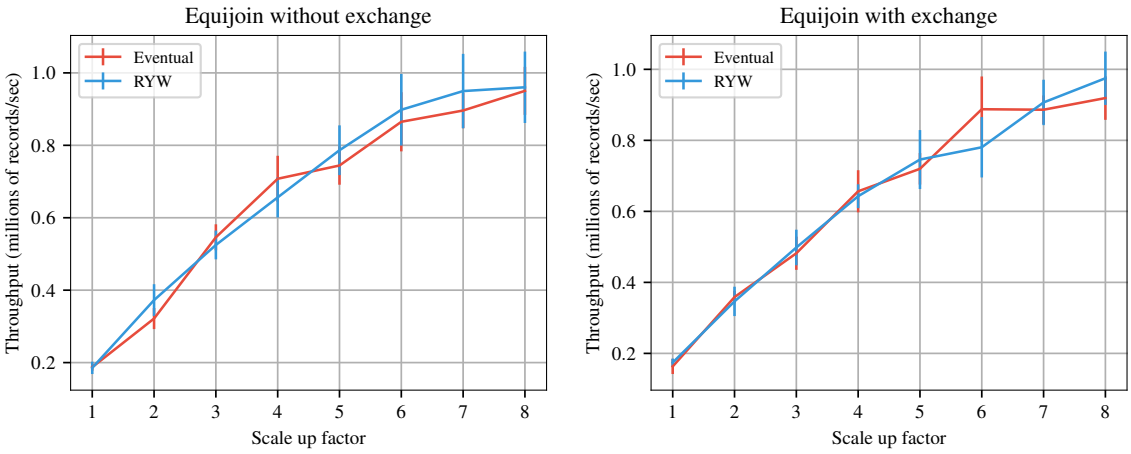
This section compares Pelton's RYW consistency mechanism with its eventually-consistent counterpart. Although this comparison is qualitatively unfair because it ignores the consistency properties, measures the overheads associated with the RYW consistency mechanism. To that end, we stress test the write path in order to measure overheads associated with write processing using a closed loop microbenchmark in section 5.1 and we measure any negative effects on latency using an open loop, realistic workload generator in section 5.2.

5.1 Write processing

In order to highlight the overheads associated with write processing, we would ideally like to have as many clients as possible issuing write requests to Pelton. However this would require significant computational resources as each client which is part of the load generator would require one thread for execution. Instead, we simulate the effect of an infinite number of clients by stashing the futures generated by write requests in a thread-local vector. This allows the system to not block on the futures while ensuring that they do not get destroyed, so that it is possible to derive and set promises associated with a particular future at a later point in time.

Since Pelton is configured to not block on the futures, in order to measure when all writes have been processed, each partitioned materialized view keeps track of how my records it has received. Once the count reaches a target value (set by the load generator), it logs the a timestamp to the console. The time to process all the records is the difference of the *maximum* of timestamps logged by *all* partitions and the start timestamp logged by the load generator. This allows us to measure the throughput for the setup where Pelton offers RYW but futures are stashed as well as Pelton with eventual consistency.

The benchmark consists of hard-coded simple dataflow graphs. It measures the overall throughput of the system while scaling the number of worker threads in Pelton. It interfaces directly with Pelton's dataflow infrastructure so that it does not measure any additional processing costs. In order to avoid overheads getting amortized with batching, the benchmark is configured with a batch



(a) (b)

Figure 5.1: System wide throughput. A scale up factor of N denotes N Pelton worker threads and N load generator threads.

size of 1. It first creates N * 100k batches and distributes them uniformly to N workload generator threads, where N is the number of worker threads in Pelton. The workload generator threads then issue write requests to Pelton. As described in section 3.3, a given input dataflow graph is also partitioned N ways. For this experiment, a single 16 core Google Cloud VM is used to execute both the Pelton backend and the load generator harness. At any point in time during the experiment there are never more than 16 software threads executing on the VM.

We compare the overall system wide throughput for two simple dataflow graphs, an equijoin graph without an exchange and an equijoin graph that contains an exchange operator (after the equijoin but before the materialized view). The input tables have basic schemas comprised of integers and the materialized views are keyed manually so that we can control to presence of exchange operators. Since the first graph does not contain an exchange, no derived promises will be required for processing. Whereas the second graph that contains an exchange, so write processing would result in lot more updates to the atomic counter as a consequence of creation and destruction of derived promises. For both of these graphs, a good result for Pelton's RYW consistency mechanism would be to have negligible overheads for write processing when compared to Pelton with eventual consistency.

Based on the results in Figure 5.1a and Figure 5.1b, the RYW consistency mechanism in Pelton does not impose any noticeable overheads for write processing in presence of large number of clients.

However, the per-client throughput with RYW enabled gets reduced (under high write load) as the client has to wait for its current write request to get processed before it can dispatch any further write requests. But this is not a meaningful metric since the overheads in this case could be easily amortized with batching, which is a sensible approach in practice if a client generates large number of requests. For the sake of completeness, we measure the throughput with the same write

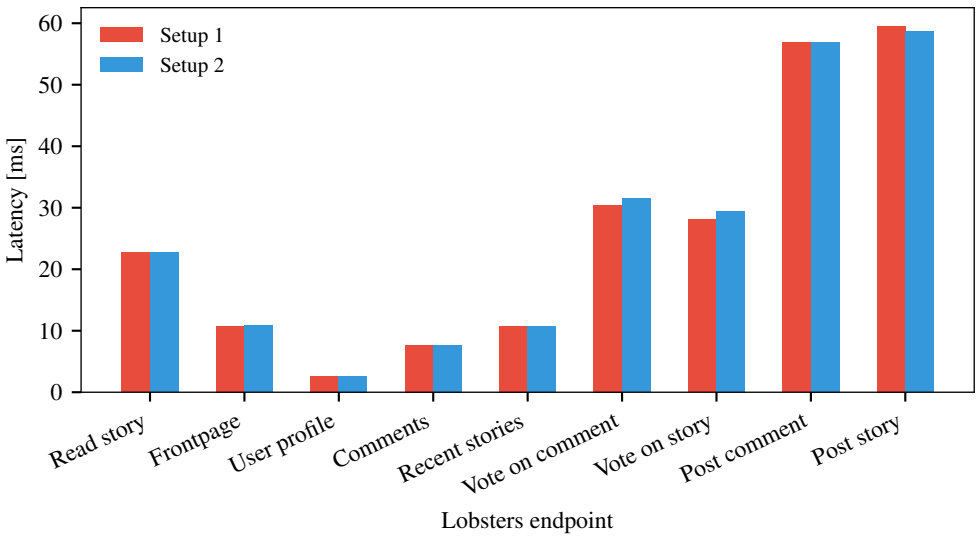


Figure 5.2: 95th percentile latencies for Lobsters application. In Setup 1, only indices provide RYW consistency whereas in Setup 2, both indices and materialized views provide RYW consistency.

benchmark and a single load generator thread acting as a client with the dataflow engine configured to block on the futures (i.e., no longer simulating an infinite number of clients). For batch size of 1, Pelton with RYW has a 5x lower throughput than Pelton with eventual consistency and for batch size of 10 Pelton with RYW has a 1.2x lower throughput than Pelton with eventual consistency. This gap closes further with larger batch sizes.

5.2 Latency

For large scale web applications, the latency observed by clients is an important metric. In order to measure the effects on latency caused by the consistency mechanism, we make use of an open loop benchmark that simulates realistic workloads experienced by the Lobsters [11] application with distributions based on metrics available in 2018 [8]. Inline with the production deployment, we populate Pelton with 5.8k users, 40k stories, 121k comments and 161k votes. The application issues 36 different types of queries, out of which 16 require views and the rest can be served directly or indirectly via secondary indices. Lobsters requires five secondary indices to be installed in Pelton, out of which one is used in order to facilitate transitive sharding and the rest are used to serve application queries.

Since the Lobsters benchmark requires Pelton's secondary indices to deliver RYW consistency, we benchmark Pelton with two configurations. In the first setup, which acts as a baseline, Pelton is configured such that only secondary indices provide RYW consistency while the remaining dataflows remain eventually consistent. In the second setup, Pelton is configured such that all dataflows provide RYW consistency. The benchmark generates a load of around 800 requests per second and measures the 95th percentile sojourn latency for all the endpoints in Lobsters. The experiment setup includes

two 16 core Google Cloud VMs with local SSDs present in the same data centre. One VM hosts the Pelton backend while the other is used to run the load generator. A good result for Pelton's RYW consistency mechanism would be that Setup 2's latencies remain comparable to that of Setup 1, especially under write heavy endpoints (vote on comment, vote on story, post comment and post story) since any overheads associated with futures and promises would be highlighted here.

Figure 5.2 describes results of the experiment. For the read heavy endpoints, there is no observable difference in latencies across the two setups. For the four write heavy endpoints, the latencies across the two setups differ by at most 1.3 milliseconds. For the Vote on Story and Vote on Comment endpoints the latency difference translates to a 3% overhead. Hence, Pelton's consistency infrastructure has acceptable latency overheads.

Chapter 6

Conclusion

This thesis demonstrates how the futures and promises paradigm can provide Read-Your-Writes consistency in an incrementally streaming computational dataflow. Even though this work was motivated by, and was completed in the context of Pelton, the techniques described here are applicable to other dataflow systems as well.

The adaptation of futures and promises in Pelton, although highly specialized, serve their purpose. They are correct, light-weight, and require little infrastructure support to integrate. Specialization isn't always necessarily bad, as often, owing to it, one can build a much more practical system.

Bibliography

[1] Kinan Dak Albab, Ishan Sharma, Justus Adam, Benjamin Kilimnik, Aaron Jeyaraj, Raj Paul, Colton Rusch, Leonhard Spiegelberg, and Malte Schwarzkopf. "Privacy-Compliant Web Applications by Construction". In: *Under submission of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Apr. 2023.

[2] *Apache Flink*. https://flink.apache.org/. Accessed: 2022-03-10.

[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. "Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 221–230.

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the USENIX Annual Technical Conference*. San Jose, California, USA, June 2013, pages 49–60.

[5] *C++ Futures*. https://en.cppreference.com/w/cpp/thread/future. Accessed: 2022-04-26.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: Amazon's Highly Available Key-Value Store". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: Association for Computing Machinery, 2007, 205–220.

[7] 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: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, California, USA, Oct. 2018, pages 213–231.

[8] 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).

[9] 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/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: Association for Computing Machinery, 2007, 59–72.

[10] Kisalaya Prasad, Avanti Patil, Heather Miller. *Futures and Promises*. 2016. URL: http://dist-prog-book.com/chapter/2/futures.html (visited on 04/26/2022).

[11] Lobsters Developers. *Lobsters News Aggregator*. Mar. 2018. URL: https://lobste.rs (visited on 03/02/2018).

[12] Malte Schwarzkopf. *The Remarkable Utility of Dataflow Computing*. 2020. URL: https://www.sigops.org/2020/the-remarkable-utility-of-dataflow-computing/ (visited on 03/10/2022).

[13] 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.

[14] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. "Naiad: a timely dataflow system". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pages 439–455.

[15] Peter Bailis. *Linearizability versus Serializability*. 2014. URL: http://www.bailis.org/blog/linearizability-versus-serializability/ (visited on 03/10/2022).

[16] RocksDB. *RocksDB*. URL: http://rocksdb.org/ (visited on 03/31/2022).

[17] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. "FlightTracker: Consistency across Read-Optimized Online Stores at Facebook". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pages 407–423.

[18] *Spark Streaming*. https://spark.apache.org/streaming/. Accessed: 2022-03-10.