University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Jan Chomiak
Student no. 394136

Wojciech Kuźmiński
Student no. 394406

Rafał Łyżwa
Student no. 394437

Jakub Martin
Student no. 394464

OctoSQL: Building a Durable SQL-Based Exactly-Once Stream Processing System With Event-Time Semantics

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
mgr Grzegorz Grudziński

Warsaw, May 2020

Abstract

Along with the recent rise in importance of data, there is a growing need for developing systems and architectures that are able to handle it in an efficient manner. The unbounded nature of many data sources like user activity on the internet or complex distributed systems has led to the creation of an abundance of streaming systems. OctoSQL is one of them and although it came to life as a batch system that allowed to run SQL queries against bounded data sources like files, relational tables and NoSQL databases, we realised that to reach its full potential it should be transformed into a streaming one. This work is an in-depth summary of the OctoSQL Streaming system and our approach to solving many, quite difficult, problems that arise when handling data in a streaming manner. Along the way we also tried to show when and how these problems arise and how ignoring them might render a streaming system a lot less useful. Apart from the theoretical examination, we also supply exemplary queries that show the usefulness of OctoSQL, as well as a brief comparison with other systems.

Keywords

SQL, Stream Processing, Exactly-Once Semantics, Event-Time Semantics,
Durable System, Databases

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

- Information systems → Data streaming; Database query processing
- Computer systems organization → Data flow architectures

Tytuł pracy w języku polskim

OctoSQL: Trwały System Przetwarzania Strumieni z Gwarancją Exactly-Once
z Użyciem SQL i Semantyką Czasu Zdarzeń

Contents

Introduction

Consider the following tasks:

- *Find the sum of all numbers in the second column of a CSV file*
- *In a JSON file of cities find the one with the largest population*
- *Consult a CSV file of cat owners and a JSON file of cats and for each cat owner find out how many cats he owns*

The first idea that might come to mind - *Easy! I will just write a short Python script for that!* So you write three Python scripts, each one solving one of the tasks. Good job! It then turns out that actually there are twenty more tasks like the three above that need to be done.

Hmmmm...

Writing twenty three Python scripts that basically do the same thing, modulo the calculation, isn't very efficient. It surely isn't very entertaining. Of course, you could write a single program that can read data from CSV and JSON files and then performs some calculations on this data. You could also load each file into a separate MySQL table and just solve the problem with SQL. But what if you need to operate on a different file each time? And each file has gigabytes of data inside? Could it be that there is no elegant, quick, reliable and easy to use solution to this simple problem?

Those were our exact thoughts when we decided to create OctoSQL for the *Software Engineering* class during the fourth semester of our Bachelor studies at the University of Warsaw. When it came to choosing the language we agreed on using Go, because of its simplicity and great library ecosystem. Although three out of four members of the team hadn't written a single line of Go code prior to the start of the semester, they quickly learned enough of the language to be able to contribute to the project. After four months of intensive work we created a batch system that could run SQL queries against CSV and JSON files, as well as data stored in MySQL, PostgreSQL and Redis databases. During the summer vacation we added Excel support, fixed bugs, introduced improvements and ran a marketing campaign on various software-related sites to get the word out about this great system of ours. We decided that OctoSQL will be an open-source project. We made the GitHub repository (github.com/cube2222/octosql) public and wrote a detailed *Readme* that would help newcomers dig into the code and write some of their own. By the start of the new semester we managed to have nearly two thousands stars on GitHub and a few people that contributed to the codebase!

And yet, we didn't rest on our laurels. We came to the conclusion that in order to reach its full potential, OctoSQL should be transformed into a *streaming* system. We contacted people in charge of the *Team programming project* (the course during which students write their

Bachelor projects and theses), who deemed such endeavour worthy of a Bachelor's degree. The following work is an in-depth description of the many brilliant and some not so brilliant ideas that make up what OctoSQL is today.

Chapter 1

Background and Motivation

In this section we describe the concepts surrounding event processing and the currently used batch and streaming designs, as well as the main hurdles that arise when building such systems.

In a typical organisation, huge amounts of data are gathered and stored in order to later make use of it in some way. One specific type of data is structured and semi-structured data - organised into records, which themselves are organised into fields or columns. Such data can describe various concepts and often we'd like to explore, analyze and aggregate it in order to achieve insights and a better understanding of it.

id	name	ownerid	age	livesleft	description
1	Buster	5	4	6	fluffy
2	Tiger	13	1	4	work of art
3	Lucy	2	7	1	fat
4	Pepper	3	14	3	work of art
5	Tiger	4	19	2	amazing
6	Molly	1	8	6	the best
7	Precious	7	16	8	lovely
8	Nala	2	10	6	cute
9	Misty	3	14	4	amazing
10	Tiger	9	20	4	true

Figure 1.1: Example record table: Cats

This data may be spread across many different locations: simple files, distributed filesystems, relational databases, document databases or event streams. That is because different types of data lend themselves to a better representation using different storage formats.

The problem with this is, you can't just query this like you could using a good old-fashioned single relational database. You have no abstraction that provides access to all this data in a single place. This is where batch and stream processing systems come into play.

1.1. Batch Processing Systems

Batch processing systems, pioneered by the original MapReduce paper [DG08] - with Apache Hadoop being widely used as an open source implementation thereof - have long been the standard way to process huge amounts of data. Systems like these are often based on dis-

tributed file systems (most notably, HDFS [SKRC10]), onto which you have to put your data prior to analysis, and which will be used as the intermediate results storage. Those usually require extensive setup to use, are however very performant. The main drawback of this is you have to process data in big batches without early results. As an example, if you're keeping track of the daily unique user count on your web page, you will get to know the count only after waiting until the next day, when complete data is available and batch processing has terminated successfully.

1.2. Stream Processing Systems

Stream processing systems aim to solve the main drawback of batch processing systems mentioned in the previous section. They process data as it becomes available, often with very small latency, making partial results possible. Two very popular stream processing systems nowadays are Apache Spark [ZCF+10] and Apache Flink [CKE+15].

However, being distributed in nature and having to run for possibly endless periods of time, stream processing systems have multiple important caveats they need to handle, some of which are described below.

1.2.1. Message Duplication

An important problem in stream processing is reliable message delivery. Those systems are usually distributed, so machines may break, networks may get severed. This may have an impact on message delivery and results in three processing models systems can support:

- at-least-once: Messages may be duplicated.
- at-most-once: Messages may be lost.
- exactly-once: Messages are processed as though there weren't any failures.

Many systems decide to provide at-least-once semantics by default, which means that events may get duplicated. This can have catastrophic consequences, as it can highly skew the output and provide a wildly different picture of the result than the truth.

Some systems, however, are able to provide exactly-once semantics using various mechanisms. A standard one is checkpointing, pioneered by Apache Flink [CEF+17], which in short, is based on making snapshots of all the stream processor state, and putting it on a durable distributed filesystem storage periodically. This has a drawback of requiring a lot of data transfer, as those snapshots often have to happen once every few seconds (record emission is committed only after a successful state snapshot). OctoSQL chooses to trust local disk storage for all data storage, to achieve performance thanks to modern SSD disks, and durability with RAID or distributed file system setups. This is described in depth in section 2.7.

1.2.2. Temporal Semantics

Another important feature of stream processing systems is Record time handling. Legacy stream processing systems treat records as if they happened the moment they've been received. However, in modern distributed systems huge message delivery delays can happen, due to outages and congestions, with records arriving hours after being emitted.

Because of this, many stream processing systems aim to provide Event-Time Semantics. With this, record time will be based on values contained in specified fields of the record. This

causes new problems to arise. When can we decide that aggregation for a given day is done? When do we know that we have already received all data there is for a given hour? Those problems can be partially solved using Watermarks, which are discussed at length in section 2.5.

1.3. Query Languages

The event processing systems described above provide SDK's to develop data analysis applications using high-level programming languages with a lot of freedom. This however means that exploratory analysis is harder and writing simple processing pipelines has more overhead than necessary. PrestoSQL [STS+19], Spark SQL [AGZ+15] and Flink SQL aim to solve this problem.

OctoSQL, too, aims to present an ergonomic SQL-based query language, extended with temporal features for event-time semantics designed for both batch and streaming use. This design was heavily inspired by [BAH+19].

1.4. Data Sources

Another important problem is, that most of the batch and streaming systems have to scan a whole dataset to use it, or even need to have the whole dataset stored on a specified distributed filesystem.

With deeper integration with the underlying databases, a data processing system can push down a lot of the filtering logic to them. A notable system designed around operation push-down is Apache Drill [HN13]. OctoSQL also tries to push down as much processing as possible to the underlying databases, in order to only download the data it really needs.

Chapter 2

Architecture and Implementation

In this chapter we describe in-depth the architecture of OctoSQL, the various tradeoffs we made and implementation details of all the notable components.

2.1. Overview

2.1.1. Query Execution

When OctoSQL starts running it parses the SQL query and transforms it into a logical plan, first of the three main intermediate representations present in OctoSQL.

The logical plan is transformed into the physical plan. During this transformation, join types are decided based on available information about the data sources as well as partition counts of future streams to be executed.

The optimiser runs through the physical plan, applying a predefined set of optimisation rules and terminates when it can't apply any more rules. The physical plan is now transformed into the execution plan with an almost one-to-one mapping of plan elements.

The execution plan can now be used to get streams from the root. This will result in streams being started and stopped in the underlying execution plan elements, producing records to the topmost stream.

This topmost stream is then directed into an output sink.

2.1.2. Record Triggering

As described before, OctoSQL makes partial results possible. Say we are counting likes for a link submission website based on a stream of reactions.

```
SELECT r.id, COUNT(*) as likes
FROM reactions r
WHERE r.type = 'like'
GROUP BY r.id
```

The stream will never end [1], so we have to decide when to send records with the current counts. One solution would be to send an updated count after each 10 received reactions. This can be simply done using the TRIGGER clause, which will be described in more depth later. Here's an example for now:

[1] Hopefully, otherwise the startup's dead!

```
SELECT r.id, COUNT(*) as likes
FROM reactions r
WHERE r.type = 'like'
GROUP BY r.id
TRIGGER COUNTING 10
```

2.2. Intermediate Representations

OctoSQL uses three main intermediate representations.

2.2.1. Logical Plan

The logical plan is a high-level description of the query in map-reduce style. Join types as well as Shuffles and stream partitions are not decided upon yet.

2.2.2. Physical Plan

The logical plan gets transformed into a physical plan. During the transformation it decides upon parallelism, join types and similar matters. An example physical plan which can be rendered using OctoSQL describe functionality can be seen in Figure 2.1

```
SELECT c.name, COUNT(*)
FROM cats c
WHERE c.livesleft > 5
GROUP BY c.name
```

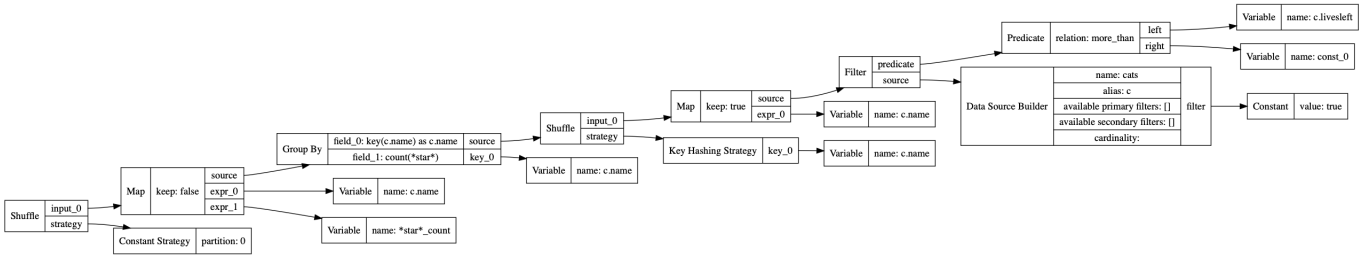


Figure 2.1: Example query with physical plan *describe* output

The physical plan already contains branches for all partitions, so we could bump up the parallelism and see it reflected in the physical plan as seen in Figure 2.2.

The physical plan exposes metadata including:

- event-time field of stream records
- cardinality of stream - is the stream finite, finite and small, or boundless

Because the physical plan gets built bottom-up, we can use the metadata of child streams to make decision regarding the creation of a parent Node, which is important, for example, when choosing between a Stream Join or Lookup Join.

Finally, the physical plan gets optimised by a rule-based optimiser.

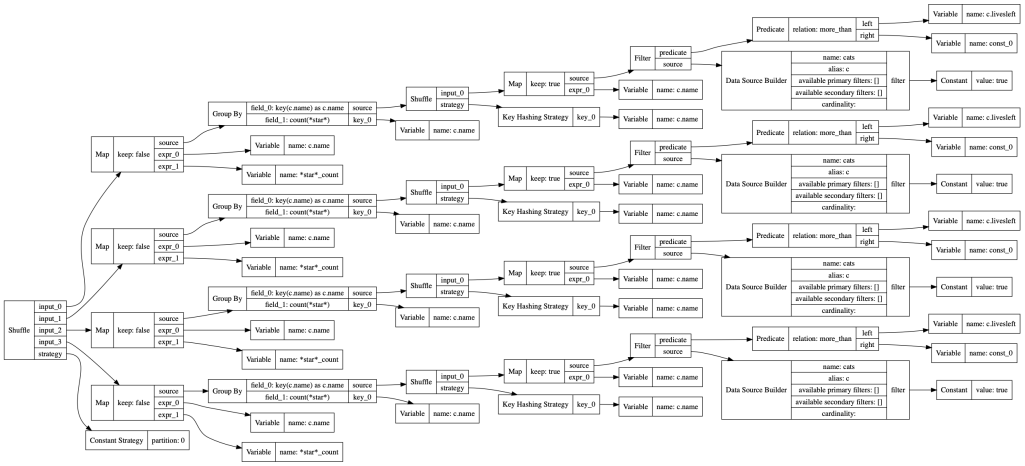


Figure 2.2: Physical plan *describe* output with groupByParallelism=4

2.2.3. Execution Plan

The physical plan gets transformed into an execution plan, which can create database connections or open files.

The execution plan is used to get a stream of records, which is also accompanied by additional metadata, including the nearest upstream Watermark Source, which is discussed in-depth in section 2.5.

The stream is now used to get new records using the *Next* method. This is similar to the model described in the Volcano paper [Gra94].

Some parts of the stream processing may be asynchronous, including streams above and below Shuffles. On top of each part of the pipeline there is a *Pull Engine*, which polls the stream for records continuously and pushes them to an *Intermediate Record Store*, which is an interface implemented by *Nodes* like Group By and will be the source of records for the next part of the pipeline.

If a stream knows it won't send any more records, it returns the *EndOfStream* message when somebody tries to call *Next* on it.

2.3. Triggers

Triggers decide when to materialise an aggregate value for a key, which is used in clauses like GROUP BY, JOIN and others.

There are four types of Triggers in OctoSQL:

- Counting
- Delay
- Watermark
- Multitrigger

2.3.1. Counting Trigger

Parameterised using a natural number n, this Trigger counts the number of records accumulated into an aggregate value and triggers once every n records.

Technically, this is accomplished using a map, which counts the number of records seen for each key in the grouping.

2.3.2. Delay Trigger

Parameterised using a delay Δ, this Trigger uses the system time t when a record for a key is received, and sets the next trigger time to be $t + \Delta$. This is achieved using a key $\rightarrow$ time mapping and a sorted (time $\times$ key) set.

When a new record for a key arrives, we can use the mappings to delete the current trigger time and replace it with a new one.

Whenever we try to send a key, we can easily check if any key triggering times are under the current system time and should be triggered, as we have the sorted (time $\times$ key) set.

2.3.3. Watermark Trigger

Let's say we want to count views per hour, and we only want the final value sent. We don't really know when all records prior to a given time t have arrived, as delays are unbounded.

Watermarks are a mechanism to approximate a time point w, which serves as a lower bound for the event times of all records to come.

The *Watermark Trigger* works similarly to the Delay Trigger described in subsection 2.3.2, with the difference, that instead of using the current system time, they use the current w.

The specifics of Watermark generation and propagation are described in section 2.5.

2.3.4. Multitrigger

We often want to receive the final value for an aggregate and partial results in-between. This is where the *Multitrigger* comes into play.

Parameterised by a group of Triggers, the Multitrigger will trigger a key if any of the underlying Triggers wants to trigger that key. Afterwards it will notify all other Triggers, that this key has been triggered. This will reset counts and times in Counting and Delay Triggers respectively.

2.4. Retraction based dataflow

However, what happens to the previous values when we trigger a key multiple times? We chose to use a retraction based model everywhere. This means that each record contains metadata which specifies - among other things - if the record is a retraction or not.

A retraction means "Forget about this record!". So whenever we trigger a new record for an aggregate, we check if we have previously triggered a value for it and if so, we send a retraction equal to the previously sent record.

As an example, say we count likes per post with a Delay Trigger. At some point in time the Delay Trigger tells us it's time to send the result we got so far. We send a record saying that a particularly funny cat meme got 15 likes. After that the meme gets more likes and now it has a thousand. When the Trigger tells us to send that value again, we say "Never mind those 15 likes, it has a 1000 likes now".

This may result in a cascade of retractions and updates downstream, resulting in a dataflow-like system.

2.5. Watermarks

All Watermark functionality in OctoSQL is inspired by the descriptions in [ABB+13], [ABC+15] and [ACL18].

When grouping records by a time window, you need to know whether all records from that time window were already received. With machine failures and congestions, extensive delivery delays could arise. Watermarks are a mechanism meant to approximate a lower bound for record event times. They are created near the data sources and propagated throughout the whole system, using a pipeline parallel to the record pipeline. A Watermark's value specifies that all records with a smaller event time should have been processed and every forthcoming record will have an event time of higher value.

Watermark Triggers work slightly differently than other Triggers. It is because the decision whether a record should be triggered is taken not purely on records event time or some fixed pattern, but on the watermark value, calculated with various strategies of the so-called *Watermark Sources*. A Watermark Source is simply an execution Node with an additional *GetWatermark* method, which returns watermark value based on calculations done during records flow. In the case when a Node doesn't affect the Watermark it uses and passes forward the Watermark Source from the underlying Node.

Examples of non trivial Watermark Sources are Shuffle Nodes, Process By Key Nodes and Watermark Generator Nodes (which will be described in more detail later on).

Shuffle Node

This Node, as covered in 2.9.6, consists of Senders and Receivers. Receivers are directly connected to the parent Node, so they need to provide a watermark value for it. How they calculate it is very simple - they just search for the smallest watermark value across all input partitions and pass it on.

Process By Key Nodes

As these processes (covered in 2.9.1) are responsible for triggering modified records, they need to manage Watermarks in a clever way so that they doesn't surpass records waiting to be triggered. This allows them to trigger records at the appropriate time. To ensure that, if there are no records waiting to be triggered, a new watermark value is set, otherwise the value is waiting in special "pending" state and it won't be raised until that happens.

Watermark Generators

The calculation methods described above are meaningful only when their underlying Watermark Sources are non nil. The first layer of the execution plan, which contains the data sources, doesn't provide any type of Watermark Source, so it passes the so-called Zero Watermark Generator - a generator that always return the zero time value. OctoSQL provides 2 types of Watermark Generators, which calculate watermark values based on raw record event times.

- **Maximal Difference Watermark Generator**

  Represents the simple approach to calculating Watermarks. Its parameters are

  - $time\_field$ - the name of the column containing event time

– $offset$ - the offset of the calculated Watermark

Given a record with event time x, it calculates $x - offset$. The Watermark passed forward equals to the maximum of $x - offset$ for all $x's$ seen so far.

E.g. with $offset = 2$ and given event times $3, 4, 1, 5, 2$, the watermark value will be 3, but if the next event time will be 7, the watermark value will change to 5.

- **Percentile Watermark Generator**

  Its parameters are

  – $time\_field$ - the name of the column containing event time
  – $events$ - represents the number of the (most recent) record event times stored (only these will be relevant in the Watermark calculation)
  – $percentile$ - represents the percentile of stored event times that must exceed the Watermark
  – $frequency$ - represents the number of records needed to start calculation process

  When a new record arrives, the oldest (according to system time of arrival, not event time!) record is forgotten, and the new one is stored and the Watermark is recalculated.

  E.g. with $events = 5$, $frequency = 5$ and $precentile = 20$ and given event times $4, 7, 2, 3, 8$, the watermark value will be 7. That's because the sorted order is $2, 3, 4, 7, 8$, $percentile = 20$ means that only 20% of records stored can be larger than current watermark value, 20% of 5 is 1 record so only 8 can exceed the watermark -> this gives us that the watermark value is 7. Now if records with event times 10 and 9 arrive, then the event times are $2, 3, 8, 10, 9$, sorted: $2, 3, 8, 9, 10$ so the watermark value is 9.

In OctoSQL late data - records whose event time surpasses the watermark value - are discarded and therefore won't be processed by the Watermark Trigger. Choosing a smaller Watermark leads to a lower probability that some late data will be ignored, but negatively impacts the performance of the system. Because of that one should be very careful when choosing a Watermark Generator.

2.6. Pull Engine

OctoSQL uses a pull-based model to send records between Nodes. For that reason, on top of each part of the pipeline we need to have an executor pulling records and pushing them to the next part of the pipeline.

This is the job of the Pull Engine. It will work the stream to completion by trying to get records from it, and pushing them into an IRS (Intermediate Record Store), which will be the source for the next part of the pipeline.

The Pull Engine works in a loop, trying to trigger batches of keys if possible, get new Watermarks and push them into the IRS, and poll the stream for records. The Pull Engine also creates and commits transactions. It pulls multiple records and/or triggers multiple keys during a single transaction. The batch size is decided using smart batching, described in section 2.6.1.

If the source stream has no records to offer yet, it will return a *Waitable Error*, which the Pull Engine will wait on, committing the transaction before. It may also return an error

indicating a new transaction is required to receive new records, which the Pull Engine also handles by committing the current transaction and creating a new one.

2.6.1. Smart Batching

Smart batching is used to decide upon the current batch size in the Pull Engine. It receives the latency target l as input, which means a transaction can't be held longer than l. This is currently set to a constant of 250 milliseconds.

The maximum batch size starts as 10. The Pull Engine will be told to finish the batch whenever we reach the maximum batch size or hit the latency target.

Whenever we successfully commit a transaction, we increase the maximum batch size slightly. Whenever we successfully commit a transaction but exceed the latency target, we slightly decrease it. Whenever we have to abort a transaction, we divide it by two.

2.7. Storage

The first wave of batch processing systems heavily relied on HDD disks and sequential access patterns. The second wave of batch/stream processing systems - led by Spark - came with the idea of using big amounts of RAM to achieve superior performance. Nowadays we have access to SSD disks which have very high throughput, even with random reads, compared to HDD disks, as well as competitive pricing compared to RAM.

This is why we decided to base OctoSQL on SSD-optimized on-disk storage. To achieve exactly-once semantics, it commits all operations as read-write transactions into an on-disk key-value datastore. We decided to use Badger as the storage engine for all processing state in OctoSQL. Badger is a fast key-value store based on the WiscKey publication [LPG+17]. It is written purely in Go, which made integrating it into our code extremely easy. Apart from that, based on benchmarks, Badger is a lot faster, when doing random reads, than the most popular key-value store - RocksDB.

In the subsections below we describe how we make use of Badger in OctoSQL.

2.7.1. Monotonically Serializable Values

Let's start with an explanation of how values are stored in our storage. Records contain various value types, from integers and strings to json-like objects, yet Badger operates only on pure bytes.

This is why every value, that OctoSQL can process, needs to be serializable into bytes. Additionally, when it comes to keys, it needs to be performed in a clever way - imagine executing an Order By when stored keys aren't sorted or using a Watermark Trigger on event times that aren't kept in an increasing order! This would add unnecessary computation to the processing of keys which would worsen OctoSQL's performance. These and many other considerations resulted in creating monotonic serialization of every value a record can contain. This monotonic serialization is used only to serialize keys that are used to store values and values themselves are stored using protobuf (described in section 2.7.5). The name stems from mathematics, where a monotonic function is one such that $x \leq y \rightarrow f(x) \leq f(y)$. This is exactly what happens during monotonic serialization. If, say, we serialize the numbers 42 and 100, then the byte sequence created for 42 will be lexicographically smaller than the one created for 100.

2.7.2. Types in OctoSQL and how they are serialized

Note: In this section when talking about "serialization" and "deserialization" we mean the monotonic version, unless specified otherwise.

In its initial version OctoSQL handled values of different types using Go's *interface{}*. For those of you not acquainted with Go, an *interface{}* can hold a value of any type, but needs to be cast to some concrete type before its value can be used. This was very cumbersome and inefficient. Moreover, such approach towards handling values of different types in statically typed languages is considered bad practice, and for good reason. Soon after the first release the type system was completely refactored, or better said, a type system was introduced.

Every value that OctoSQL can handle is of the "supertype" *octosql.Value*, which is a proto generated structure. Its subtypes are:

- Null
- Phantom
- Int
- Float
- Bool
- String
- Timestamp
- Duration
- Tuple
- Object

We will now briefly discuss how these types are serialized and deserialized. Each type in OctoSQL has an integer identifier. When a value of any type is serialized this identifier is placed at the beginning of the byte array, so that the value can be correctly deserialized later. This could lead to problems when lexicographically comparing the bytes resulting from serializing a tuple of type [float, int, duration] and a tuple of type [int, duration, float], since their order would only depend on the fact that int's identifier is smaller than float's. This is not a problem, however, since values of different types should never be stored in the same data structure.

Null, Phantom, Bool

As mentioned above, each type in OctoSQL has its unique identifier. The serialization of Null and Phantom is just returning the appropriate identifier as a byte. The serialization of Bool is two bytes: the identifier and the boolean value itself. It is easy to see how one would deserialize each of these types.

Int, Duration, Timestamp

Let's say we want to serialize an integer value i. Serialization of an int results in a byte array of length 9. The first byte is the int identifier. The next 8 bytes are the big-endian representation of the value $uint64(i \wedge math.MinInt64)$, where $\wedge$ is the XOR operator. Since the big-endian representation of $math.MinInt64$ is [128 0 0 0 0 0 0 0], what this does is simply switching the sign bit of our integer. This way non-negative integers have the first bit of the big-endian representation set to 1 and the negative ones to 0. Thanks to that we automatically get the fact that the serialization of any negative number will be lexicographically smaller than the serialization of any non-negative number.

We will start with negative numbers. They are stored using the two's complement, which means that the representation of a negative number n is the number $2^{63} + n$. Because of that, if we have two negative numbers $x \leq y$, then $2^{63} + x \leq 2^{63} + y$ - the larger number has a larger two's complement. Since the two's complement is a positive number we just need to make sure that the big-endian representation of positive numbers is monotonic. It is easy to see that it is: in the big-endian representation (of 64-bit integers) the leftmost bit (excluding the sign bit) represents the power 2^{63} and the rightmost bit represents 2^0. If we have any two positive integers x and y, such that $x < y$, then they are equal on the first m (possibly 0) bits and then the $(m+1)$-st bit of x is 0 and that of y is 1. This fact shows that the serialization of x will be lexicographically smaller than that of y

The duration type is serialized using the int serialization, since in Go this type is just $int64$.

To serialize a timestamp value we take its Unix epoch time and the number of nanoseconds after the full second, we serialize both values using the int serialization and in the end the result is the timestamp identifier and the two serialized values concatenated (seconds first, then nanoseconds).

String

The biggest problem with strings is their varying length. Because of that there needs to exist some value that will signal that we have reached the end of the string. It cannot appear in the serialization of a string, because then we would wrongly say that the string has already ended. To avoid this, we serialize strings like this:

1. Put the string identifier at the beginning of the array
2. Cast the string to a byte array
3. Split each byte from this array into two bytes, so that each byte has value ≥ 128. This is achieved by writing each byte in base-128 and then increasing both positions by 128.
4. Append the string delimiter (11) at the end of the array.

This way when deserializing a string we know that if we see a byte value 11, then it is the end of the string and not some value describing the next character in it.

Float

The code for serializing float values was based on
https://github.com/danburkert/bytekey/blob/master/src/encoder.rs, which bases it implementation on information from [HSW13], section 17-3.

Tuple

Serializing a tuple is the easy part. The first byte is the tuple identifier. Then just iterate over the values in a tuple, serialize them and append the resulting bytes to the serialization. Once that's done, append the tuple delimiter at the end of the byte array. The hard part is taking an array of bytes that represent a tuple and recreating it. For previous types we didn't really talk much about deserializing values, because the operations that were performed were easily reversible. The problem with deserializing a tuple is the fact, that it might contain some other tuple as one of its elements. To solve it a recursive method is used, that returns the number of bytes it read in order to deserialize some value:

1. Set an index to point at the start of the tuple (the first byte after the identifier)
2. Check the value of the byte at the index:
   (a) If it's the tuple delimiter we have reached the end of the tuple and we can return the result (an array of all values deserialized up to this point)
   (b) If it's an identifier of a type with constant length serialization (i.e. int, null etc.), deserialize it and return the number of bytes read along with the deserialized value
   (c) If it's an identifier of a string, deserialize it reading bytes until the string delimiter appears and return how many bytes were read along with the deserialized string
   (d) If it's an identifier of either a tuple or an object, call the function recursively and return the result of that call
3. Once you have the number of bytes read in that step and the deserialized value, append the value to the result, move the index to the right by the number of bytes that were processed and go to point 2).

Object

The object type in OctoSQL is a map. An object is serialized by transforming it into a tuple, who's first n entries are the keys of the object, and the next n are the values corresponding to those keys (where n is the number of keys in the object). Since an object is serialized as a tuple, it is also deserialized as a tuple, with the only addition that after deserializing the tuple is transformed back into an object.

2.7.3. Data Structures

To prevent us from operating on raw Badger transactions, we created a few data structures similar to the ones known from the high-level programming languages. They provide high-level methods and communicate with the Badger database underneath, which makes them a comfortable Badger overlay. The data structures are on-disk imitations of the equivalent in-memory structures. The only parameter they require is a suitably prefixed Badger transaction.

OctoSQL uses four different Data Structures:

- ValueState - this is the simplest data structure which allows storing and reading a singleton value. It is useful, for example, for storing an aggregated sum, a watermark value, a datasource offset or a record count. Methods provided by ValueState are: *Get(value)*, *Set(value)*, *Clear()*

- MultiSet - originally created as the standard Set, it quickly became the multi-key version. It uses a hash function to separate stored values. Methods provided by MultiSet are *GetCount(value)*, *Insert(value)*, *Erase(value)*, *Contains(value)*, *Clear()*, *ReadAll()* and *GetIterator()* (with methods *Next(value)* and *Close()*).

- Map - this is the only data structure that is based on monotonically serialized keys, so apart from standard functionalities, it stores keys in the order of their byte representation, which is particularily useful in handling records' event times. In contrast to MultiSet, Map's methods mostly take 2 arguments, *key* and *value*, where value read from the storage is saved to *value*'s pointer. Methods provided by Map are *Set(key, value)*, *Get(key, value)*, *Delete(key)*, *Clear()* and *GetIterator()* (with methods *Next(key, value)* and *Close()*).

- Deque - originally created as a standard List, it quickly became the double-ended version. It is useful for example in first/last aggregates or record queues (used whenever one pipeline part sends data to another one) Methods provided by Deque are *PushFront(value)*, *PushBack(value)*, *PopFront(value)*, *PopBack(value)*, *PeekFront(value)*, *PeekBack(value)*, *Length()*, *Clear()* and *GetIterator()* (with methods *Next(value)* and *Close()*).

2.7.4. Subscriptions

The disk storage in OctoSQL provides subscriptions. Subscriptions allow somebody to wait until a given data structure changes in any way. We also provide subscription concatenation functionality, which makes it possible to wait on multiple data structures simultaneously.

This has been very straightforward to implement thanks to Go channels.

2.7.5. Serialization

Data structures in OctoSQL can store values serializable as a Protocol Buffer. This resulted in records, all types of values, and all states that Nodes may want to contain being specified as protobuf messages.

2.8. Restart Durability

The main reason behind keeping all state on disk is to provide restart durability. In order to achieve this, the state of each Node is stored in key-value pairs, with the keys prefixed using the Nodes *StreamID*.

Whenever a Node starts a stream from a child Node, it allocates a StreamID and saves it in its own stored state with an input ID. If a StreamID already exists for that input ID, the existing one will be used. The stream created from the root Node gets a predefined StreamID - *root*.

This way, when OctoSQL is restarted, each stream will recursively be created with the same StreamID it had on the previous OctoSQL run with all state kept in place.

2.8.1. Transaction namespaces

Almost every part of the execution plan uses an on-disk storage to save current calculations, states, record values and other elements. Several of them need many data structures at the same time. How is this all handled to prevent transaction and database key collisions?

Let's start by saying, that a transaction can hold a *prefix*, which is a slice of bytes. The prefix indicates a namespace for the structure using this transaction. Transactions can be used to get nested ones prefixed by an additional byte sequence.

Also, as said in the 2.8 section, every Node gets a unique StreamID, which can be used as a prefix. This is mostly to avoid key collisions between Nodes and also allows us to clear all the storage of the Node easily just by dropping this prefix from the database.

So now we know that transactions can have a prefix, and every Node prefixes their data structures with its StreamID. But how can we separate the data structure namespaces if the transaction still isn't distinguished between the structures? The answer is very simple: we can just append another prefix, this time predefined for every data structure used!

To resolve any remaining doubts, imagine having a Map prefixed with *prefix1* and creating another one prefixed with *prefix2*. Now the whole state of the second Map will have different keys thanks to the different prefix. This means that you can perform any actions on the second Map without interfering into the namespace of the first one.

2.9. Node Types

Below we describe the interesting Nodes that exist in OctoSQL. What exactly do we mean by interesting? A *Map* or a *Filter* aren't really interesting. Better said: their streaming versions don't differ much from their original batch counterparts. A Map takes a record from its source, modifies it and sends it through. A Filter takes a record from its source, checks whether the filter predicate holds for that record and based on that sends it through or not. Nothing is persisted to disk, nothing fancy happens with any transactions. It's basically the same as in the original OctoSQL.

Before we go further, we must know what a Node actually is. *Node* in OctoSQL is a single-function interface:

```
/* arguments and errors omitted for simplicty */
type Node interface {
    Get(...) RecordStream
}
```

Ok, so a Node is just something that, when given the right arguments, gives us a stream of records. Simple enough. With that knowledge we are ready to dive right into the world of the Nodes of OctoSQL:

2.9.1. Process By Key and Process Function

And to start things off we get the *Process By Key* structure and the *Process Function* interface, neither of which is really a Node. Now that's a bummer... Don't worry, they are pretty cool too, and what's even more important, they greatly simplified writing the *Group By*, *Stream Join* and *Distinct* Nodes, so don't quit on them just yet.

We will start with Process Function, because it's the simpler of the two. As mentioned above, it's an interface:

```
/* some arguments and errors ommited for simplicity */
type ProcessFunction interface {
```

```
    AddRecord(inputIndex int, key octosql.Value, record *Record)
    Trigger(key octosql.Value) []*Record
}
```

This is pretty straightforward, but to quote everybody's all time favourite: *Everything should be made as simple as possible, but not simpler.*[2] And that's exactly what Process Function achieves. It can receive a record and include it in its calculations (all information that it gets is the key of the record, the record itself and from which stream the record came) and return all records that result from those calculations for some key. Later on we will see that Group By, Stream Join and Distinct all implement the Process Function interface. As you read on it will become clear why, but for now it suffices to see that they perform calculations on groups of records that are separated into buckets based on their key (what the key is for each one will be explained in the appropriate section).

OK. With Process Function out of the way we will try to tackle the Process By Key structure. Before diving into its details we will try to sell some intuition behind what it's for. In OctoSQL, and more generally in streaming systems, there is a lot of work concerning the handling, storing, updating and pushing down Watermarks, Triggers and other things that are "the metadata of the pipeline". This work doesn't depend on the actual calculation being performed. From a Trigger's point of view it doesn't matter if it tells a Group By or a Stream Join to trigger some key or if that message is handled by some other Node, that eventually tells the Group By to trigger a key.

It also makes sense that we wouldn't want to write Watermark handling and passing around for the Group By, Stream Join and Distinct Nodes separately, although the implementation would be exactly the same. We would want to separate the calculations from the "pipeline metadata" handling. And that's exactly what the Process By Key structure does. It handles all the "dirty work" and allows the Process Function to take care of the calculations.

Alright, let's take a look inside of this bad boy, then:

```
/* order of fields changed */
type ProcessByKey struct {
    processFunction ProcessFunction
    trigger Trigger
    keyExpressions [][]Expression
    eventTimeField octosql.VariableName
    variables octosql.Variables
}
```

The last two fields aren't really that important to understand what's going on, so we will just touch upon them briefly. The *eventTimeField* describes the name of the column that contains the event time of records, if we are "grouping" by event time i.e. the event time forms part of the key. The *variables* are just some variables that are currently in scope. Now onto the important stuff.

First, we have a *processFunction* that represents the actual calculation, for example a Group By. The *trigger* is just the underlying Trigger that passes us information about when to trigger values. When the Process By Key correctly receives and processes some record, it

[2] Many attribute these words to Albert Einstein, but upon deeper inspection it is not entirely clear whose quote that is.

informs the trigger about this fact. We mentioned before that we want a Process Function to perform calculations on buckets of records and that the segregation into buckets is based on something. That's exactly the duty of the *keyExpression*. The first index in this array is just the index of the source. It is necessary in a Stream Join because the two source streams usually have different key expressions. Say we have just a single source and the keyExpression[0] looks like this: $[p.\text{city}, p.\text{age} * 2]$. When we receive a record we just calculate all the values from the appropriate keyExpression and get an array of values like ['Warsaw', 42]. We then create a tuple from this array and that's the key for this record. Now when passing this record to the Process Function we will also pass the key along with it and that will allow the Process Function to place it in the appropriate "bucket" with other records.

2.9.2. Group By

Before addresing the main topic of this section we will first introduce the notion of an *aggregate* since it carries out group by calculations. It is also a good practical example of modifying and mantaining state in a streaming system. In OctoSQL, **Aggregate** is an interface that supports three operations: adding a value, retracting a value and returning the value aggregated so far.

Below we will shortly describe each aggregate that is implemented in OctoSQL, starting with the simpler ones:

COUNT

The most basic aggregate that just keeps track of the number of values it has seen. It consists of a single ValueState in which the counter is stored.

- Adding a value - increase the counter by 1
- Retracting a value - decrease the counter by 1
- Get aggregated value - return the counter value

It is of educational value to mention how an implementation of such a mechanism differs when operating in memory and on disk. If the counter were stored in memory we would simply do with a structure that has a field *counter* and three methods that essentialy are counter++, counter−− and return counter. In the case of an on-disk counter it first needs to read the octosql.Value stored in the ValueState, check for errors, interpret the value as int, increase/decrease it by 1, convert it back to octosql.Value, store it back on disk and check for errors again.

SUM

Another fairly easy aggregate, which keeps track of the sum of the values it has seen. The main difference, compared to the Count aggregate is that we need to distinguish value types (one can only sum integers, floats and durations) so an additional ValueState that stores the number of records seen is needed. This is because when this number is greater than 0, any new value must match the type of the sum aggregated so far and in the other case, every value is acceptable.

- Adding a value - if the value is the first one seen (counter = 0) then set this value, otherwise if both types match then set "aggregated + value"; increase the counter
- Retracting a value - if both types match then set "aggregated - value" and decrease the counter

- Get aggregated value - return the aggregated sum

AVG

An aggregate that uses an underlying Sum and Count, used to calculate the average of values it has seen. Just like with Sum, only integers, floats and durations are acceptable.

- Adding a value - call *AddValue* for both underlying aggregates
- Retracting a value - call *RetractValue* for both underlying aggregates
- Get aggregated value - get values for both underlying aggregates and return $\frac{aggregated\ sum}{aggregated\ count}$

DISTINCT (COUNT/SUM/AVG)

A slightly more complex example, as it uses a Map instead of a ValueState, this aggregate requires an underlying Count, Sum or Avg. It's used to calculate the value of the wrapped aggregate, but takes only one copy of every value into consideration. The Map takes the form of $value \rightarrow its\_counter$ and it's used to delete value from it whenever the number of value additions equals the number of its retractions.

- Adding a value - if the value already exists in the Map then increase its counter, otherwise add a new mapping of $(value \rightarrow 1)$ AND add the value to the underlying aggregate
- Retracting a value - if the value already exists in the Map and its counter equals 1 remove the value from the Map AND retract the value from the underlying aggregate, otherwise decrease its counter
- Get aggregated value - get aggregated value from the underlying aggregate

Notice that every action is taken only when the incoming value wasn't present in the Map (adding) or was present once (retracting), which means that the aggregated value is calculated based on one copy of every given element.

MIN

This aggregate keeps track of the smallest of the values it has seen. It uses a Map for the same purposes as Distinct does - to keep track of the values' counters.

- Adding a value - if the value already exists in the Map then increase its counter, otherwise add a new mapping of $(value \rightarrow 1)$
- Retracting a value - if the value already exists in the Map and its counter is equal to 1 remove the value from the Map, otherwise decrease its counter
- Get aggregated value - get an iterator for the Map and return the result of the *Next* method. Notice that keys in Map are stored in an increasing order, which was stated in subsection 2.7.1, so calling *Next* gives us the smallest value! Additionally, if the smallest value was both added and retracted, it must have been deleted from the Map so a retracted value cannot be the result of the Min aggregate.

MAX

Twin to the Min aggregate, the one and only difference is that instead of getting a Map iterator in *Get* method, it creates a reversed iterator. As a result, one call of *Next* iterator method returns the biggest of the values instead of the smallest.

FIRST

Definitely the most complex aggregate, which keeps track of the first of values it has seen. It uses both Deque and Map to do its duty. The Deque is responsible for keeping values in appearance order while the Map is used in the same way as in Min/Max aggregates.

- Adding a value - if the value already exists in the Map then increase its counter, otherwise add a new mapping of ($value \rightarrow 1$); always *PushBack* the value to the Deque
- Retracting a value - if the value already exists in the Map and its counter is equal to 1 remove the value from the Map, otherwise decrease its counter; don't perform any actions on the Deque
- Get aggregated value - iterate through the Deque - this can be done by using *PeekFront* and *PopFront* combination. For every element found, check its counter in the Map. If it's positive, then the value wasn't retracted and therefore indicates the first non-retracted element (which is exactly what we're looking for) - return the element and stop iterating. Otherwise, pop the value from the Deque - it must have beed already retracted and thus shouldn't be considered as an existing value. Notice that combination of *PushBack* and *PopFront* forms a queue in a FIFO context.

LAST

As with the Min/Max similarity, Last can be considered as a twin brother of First. The only difference is that instead of *PushBack* while adding a value, it uses *PushFront*, but the iterating process is still the same (*PeekFront* + *PopFront*). that combination of *PushFront* and *PopFront* forms a queue in a FILO context, so the first non-retracted value is the last element seen (which is exactly what we're looking for).

KEY

Mentioned in the end as it is the most specific aggregate, it is used to remember Group By's current key. It uses 2 ValueStates - one to store the actual key and one to remember its counter (just like the Count aggregate does).

- Adding a value - if the first ValueState is empty, then store the value and set the counter to 1, otherwise increase the counter
- Retracting a value - if the value already exists in the ValueState and its counter equals 1 clear both ValueStates, otherwise decrease the counter
- Get aggregated value - return the value stored in the first ValueState if the counter is positive (which indicates that the key is still valid)

Back to Group By

Now back to the topic at hand! A Group By in OctoSQL is a Process Function and runs using the Process By Key abstraction (explained in section 2.9.1). Say we run a group by query against a table *people*:

```
SELECT p.city, COUNT(*), AVG(p.height), MAX(p.cats_owned)
FROM people p
GROUP BY p.city
```

In a Group By the two most important things are **the aggregates** and **the key**.

Aggregates for a Group By are the function-like names that appear after *SELECT* (some of them are implied, like the *KEY* aggregate for p.city here). In this query we have four aggregates: *KEY*, *COUNT*, *AVG* and *MAX*.

The key, as in all Process By Key instances, makes sure that records that should be aggregated together, are aggregated together, and that records that shouldn't, are not. In the case of the Group By the key is an array of expressions that appear after the **GROUP BY** clause in the query. In our example this array would be [p.city].

So how does the Group By work in terms of implementing the Process Function interface? It's fairly straightforward. From section 2.9.1 we know that each Process Function supports two methods: adding a record for some key and triggering a given key. To explain how this works, let's consider those two simplified code snippets:

```
/* error handling omitted for simplicity */
func (gb *GroupBy) AddRecord(tx Transaction,
                             key octosql.Value,
                             record *Record) {
    for i := range gb.aggregates {
        value := record.Value(gb.inputFields[i])
        prefixedTx := tx.WithPrefixes(i, key)
        if !record.IsUndo() {
            gb.aggregates[i].AddValue(prefixedTx, values[i])
        } else {
            gb.aggregates[i].RetractValue(prefixedTx, values[i])
        }
    }
}

func (gb *GroupBy) Trigger(tx Transaction, key octosql.Value) []*Record {
    values := make([]octosql.Value, len(gb.outputFields)
    for i := range values {
        prefixedTx := tx.WithPrefixes(i, key)
    values[i] = gb.aggregates[i].GetValue(prefixedTx)
    }

    /* handle creation of record and sending possible retraction */
}
```

In the AddRecord method we simply get the value meant for each aggregate (in our query grouping by city it would be: the city, the whole record, the height and the number of cats owned) and pass it to the aggregate (taking into consideration whether we are adding or retracting it). The Trigger method just gets the value from each aggregate and creates a new record based on those values. It is important to remember that if some key was already triggered, then along with the new value for it, we must retract the previous one. Both functions are in reality a bit more complicated: they handle deciding which field is the event time field, assign names to columns in the new record, handle errors, and do a bunch of other stuff. Nonetheless this piece of code is enough to understand the main idea behind the Group By.

2.9.3. Distinct

The *Distinct* Node also implements the Process Function interface and uses the Process By Key abstraction. Luckily enough, it is way simpler than the two Nodes that do: the Group By and the Stream Join. When thinking in terms of the Process By Key we know we want to group together records that have equal keys, because records with equal keys should be aggregated together. So what should the key be for the Distinct Node?

In OctoSQL Distinct works like this:

- When a record arrives, increase its count by 1. If that's the first time such record has appeared, store it, so it can be triggered later.
- When a retraction arrives, decrease the count of the retracted record by 1. If the count is greater than 0 after that, that's it. Otherwise, if this record was already triggered (it's not present in the store from the previous point) we send a retraction for it. If not we just clear the store.

This can be implemented quite easily using the ValueState structure. It is clear that in Distinct we want the key to be the record's columns and values. We wouldn't want to include ID because, well, this way all records would be different from one another. We also don't want to include the information whether a record is a retraction or not, because as described above, retractions should end up in the same bucket as the record they are retracting. The last thing one could consider is whether two records could have the same column names and values, but different columns describing their event time. In theory that's possible, but we decided to assume that it would be a bit weird, so in the end this information isn't included in the key.

Since during the creation of the logical plan we don't know what the column names of the records will be in our data source we use the *RecordExpression* as the key. It is simply an expression that, when given a record, returns a tuple that consists of its column names and values.

The implementation of Distinct also comes with a story. At first we said that, in SQL, **DISTINCT** is just **GROUP BY *** and so we tried to implement Distinct using the already implemented Group By. The main argument for that was the fact that some code would be the same in both Nodes, so there was no point in rewriting it. This resulted in complicated code, which was faulty and not resilient to restarts. After many hours of trying to fix it, we decided to just rewrite Distinct using the Process By Key abstraction. That was done within a day and worked on the first try. We think the moral of the story speaks for itself.

2.9.4. Stream Join

Note: in this section the part of a join query after the keyword **ON** will be called a *predicate* or a *join predicate*.

OctoSQL supports two types of joins: a *Stream Join* and a *Lookup Join* (explained in section 2.9.5). This distinction isn't visible to the user - the type of the join is decided upon the underlying streams and the predicate of the query. The Stream Join is the easier (in terms of the logic behind it) and usually the faster of the two, but it comes with some limitations.

The Stream Join implements the *ProcessFunction* interface and works in the *ProcessByKey* abstraction (both explained in section 2.9.1), thus, exactly like in a Group By, we want any two records with the same key to be processed together, and any two records with different keys to be processed separately. Remember that we are joining two streams, so two records are matched only if they come from two different streams and they have the same key. A natural idea is to create the key based on the predicate. Let's look at an example:

```
SELECT *
FROM people p1 INNER JOIN people p2
ON
p1.age = p2.age + 5 AND
p1.city = p2.city AND
p1.first_name = p2.second_name
```

We match two people if and only if the first is 5 years older than the second one, both are from the same city and the first person's first name is the second person's second name. In this case it is easy to create a key for both streams. For the first stream it will be an array [p1.age, p1.city, p1.first_name] and for the second one: [p2.age + 5, p2.city, p2.second_name]. This way if we get the record:

```
{
    p2.age = 23,
    p2.city = 'Warsaw',
    p2.first_name = 'John',
    p2.second_name = 'James'
}
```

from the second stream, we will calculate that its key is [28, 'Warsaw', 'James'] and we will send that value to the Stream Join along with the index of the stream (2 in this case) and the record itself. For any record from the first stream to match with this record it will need to have the exact same key, and that's exactly what we're aiming for.

So what does a Stream Join do under the hood? It keeps track of four sets of records for every key: "old" (ones that already were present during some trigger in the past) records from the first stream, "old" records from the second stream, "new" (ones that were added after the last trigger) records from the first stream and "new" records from the second stream. To simplify explanations we will call them *OldFirst*, *OldSecond*, *NewFirst* and *NewSecond* respectively. When a record arrives it is simply added to the appropriate "new" set. The matching of records is carried out, when a key is triggered. The new matches are: *OldFirst* × *NewSecond*, *OldSecond* × *NewFirst* and *NewFirst* × *NewSecond*. After the matches are created, records from the "new" sets are moved to the appropriate "old" set, and the "new" sets are cleared. Matching (or merging) two records together is just creating a new record

that has columns from both the records and the values as the values in the records. This new record can be a retraction, if either of the two records that are matched is a retraction.

It is important to mention how a Stream Join handles retractions. A count is kept for every record (two records are included in the same count if they have equal fields and columns). A retraction decreases the count by 1. A standard record increases it by 1. This way early retractions aren't a problem - the count drops to some value below zero, and then along with the arrival of normal records gets bumped up back to zero and then to some positive value. When a retraction for a record arrives there are two possibilities:

1. The record that's retracted is still in one of the "new" sets. This means that it hasn't been triggered yet, so we can just remove it from the set.

2. The record is not present in a "new" set. This means that it was already triggered so we have to inform the rest of the system to ignore it. This is achieved by inserting the retraction into the appropriate "new" set. This way when matching records in *Trigger* method, every record that was formed thanks to the retracted record will be retracted itself.

To finish this subsection we will touch upon the aforementioned limitations of the Stream Join. They stem from the fact that we are operating in the ProcessByKey abstraction. It is easy to see that a predicate like this one:

```
... ON p1.age < p2.age OR p1.city <> p2.city
```

can't be handled using ProcessByKey. We have no way of assigning a key to records from both streams, such that equal keys mean they should be matched in the join. Even if we had only equalities, but used an **OR** somewhere, it still wouldn't work. Thus a Stream Join is limited to queries, of which the predicate is a conjunction of equalities.

2.9.5. Lookup Join

The Lookup Join handles all queries the Stream Join cannot. It has a strict separation of the *source* **stream** and the *joined* **Node**, which are handled completely differently, as opposed to how the Stream Join handles inputs.

The Lookup Join will pull records from the source stream. For each record it will start an asynchronous task, which starts a joined **stream** for this record (which satisfies all the join predicates using a Filter Node). The joined **stream** will return all records that should be matched with this source record. For each new record we get from the joined stream, we send a merged record based on the source and joined record downstream.

Let's look at this in more detail now. The Lookup Join itself is an Intermediate Record Store, as described in section 2.6. Whenever it receives a record from the Pull Engine, it will be put into a queue to be processed. An asynchronous task - the *scheduler* - reads records to be processed, creates a job entry in a *jobs* map, gets a stream from the joined Node using the source record and starts a *worker*, which will drive this stream to completion. Each joined record will be put into a job-local output records queue. When the stream is done, an *EndOfStream* entry is appended to this queue.

The scheduler makes sure to only run up to *lookupJoinPrefetchCount* asynchronous workers at the same time.

When the Lookup Join is polled for records, it will iterate through the *jobs* map, and for each job it will check if there are any output records queued. The first encountered record will

be merged with the given job's source record and returned. In case an *EndOfStream* message is encountered, the job entry will be deleted and its entire local state cleaned up.

Whenever OctoSQL is restarted, a worker task is started for each existing job entry to resume in-progress join execution.

A token queue is used to provide back-pressure. For each record written to the Lookup Join to-be-processed queue, a token has to be consumed. Every time the Lookup Join finishes a task, it will produce a new token. Limited initial tokens are produced when the Lookup Join is created to get started.

2.9.6. Shuffle

Before we discuss how the Shuffle Node works, we'll go over the multi-partition nature of OctoSQL. When the physical plan is created from the logical plan, in some cases logical Nodes will return a list of physical Nodes, one for each partition.

Trivial Nodes, such as Map or Filter, which are *stateless* and only look at one record at a time, will just create one instance of themselves for each source partition. In other words, if the child of some logical *Map* Node returned four partitions during its transformation into the physical form, then the Map will create one instance of a physical Map for each one of the child partitions and then return these four instances.

However, there are Nodes which work on multiple records at a time, i.e. Group By, which need to have all records with a given key in a single partition, so that they can find their way into one aggregate and be combined.

In order to put all the records with the same key into a single partition the *Shuffle* Node is used.

A physical Shuffle Node gets multiple input partitions and returns a specified number of output partitions - which can be parameterised, for example using the *groupByParallelism* configuration variable for Shuffles preceding a Group By.

Each Shuffle Node will materialize into a single Shuffle execution Node.

When we start the root streams, we will pass special *ShuffleID*'s down, which will be further passed down recursively, possibly changed if a Node has multiple different children, a *Union All* Node for example, so that each child gets a unique one. This will later be returned back up as a mapping of ShuffleID $\rightarrow$ Shuffle Instance.

The Shuffles below have actually been turned into *Shuffle Receivers* - stream Nodes which receive records from a Shuffle - during the physical to execution plan transformation, with the ShuffleID they've received. Now we start the streams beneath the Shuffle the same way we started the root stream and proceed in a recursive fashion.

Shuffle Mechanics

A Shuffle works by using *input partitions* $\times$ *output partitions* queues. Each input partition has a *Shuffle Sender* at its root with a Pull Engine driving it to completion. Each record gets sent into the appropriate output partition queue. We use this many queues, so that each Receiver has one queue per Sender and the Senders don't interfere with each other.

Shuffle Strategies

The output partition for a record is chosen using a shuffle strategy. The available shuffle strategies are:

- Constant - always sends to the same output partition. This is useful if we only have a single output partition and want all records to end up there.

- Key Hashing - use $Hash(key)$ modulo *output partitions* as the output partition. This causes all records with the same key to be sent to the same output partition and partitions all keys more or less evenly.

2.9.7. Order By

Let's look at this Order By query against a table *cities*:

```
SELECT c.name
FROM cities c
ORDER BY c.country ASC, c.population DESC
```

In an Order By we have **the key** created from expressions and order directions.

Expressions represent fields in a record by which we sort. For each expression there is one order direction. Order directions (*ASC*, *DESC*) represent ascending and descending orders. The order of query expressions corresponds to the order of variables used for sorting.

In this query we have two expressions: *c.country*, *c.population* with two corresponding order directions: *ASC*, *DESC*

An important thing to remember is the fact, that an Order By usually needs to store all the values it has seen, in order to function properly. It needs to store all values that have not been triggered yet and it can't trigger the first value, before receiving the last, because the last received value could be the first in the chosen order The only exception to this is when the first value that the query orders by is *event time asc*. When it happens, Order By needs to only remember values that haven't been triggered yet, and values can be triggered based on event time.

An Order By Node uses on-disk data structures to provide correct key ordering.

So how does an Order By Node work?

For storage it uses a Map $key \rightarrow pair(record, counter)$. There is one Map for each prefix. Prefixes are passed by transaction and contain an event time when the first value that a query orders by is *event time asc* and a specified constant value otherwise.

When a new record arrives we calculate its key based on the query expressions and the values in the record. The keys are comparable and the function that maps records to keys is monotonically increasing. To create the key an Order By Node iterates over pairs(expression, direction). For each pair the Node monotonically serializes the expression value of the record and if the direction is decreasing it applies (xor 255) to every byte in the serialized value to reverse the order of direction. The key is a concatenation of these values and the record serialized, separated by a special character. Then we update the counter in the Map corresponding to the received record.

Finally, when the time comes, the values are triggered. The Order By iterates over the Map for the triggered prefix and sends records down the pipeline.

2.9.8. Tumble

A *Tumble* is used to append time windows to the records based on their event time. It needs three parameters:

- $time\_field$ - the name of the column containing event time
- $window\_length$ - the length of the time window assigned
- $offset$ - the offset of time windows

Both $window\_length$ and $offset$ are represented with time durations. Every window will have a form of $[x, x + window\_length]$, where $x =$ (some multiple of $window\_length$) + $offset$.

Take a look at this simple example: assume $window\_length = 10$ minutes and given event times 11:02, 11:13, 11:27, 11:41 the time windows will be

- for $offset = 3$ minutes: [10:53, 11:03], [11:13, 11:23], [11:23, 11:33], [11:33, 11:43]
- for $offset = 1$ minute: [11:01, 11:11], [11:11, 11:21], [11:21, 11:31], [11:41, 11:51]

Whenever a Tumble pulls a record from the underlying source, it appends two additional fields to it: the start and the end of the appropriate time window. Then it sets the end of the window as the record's event time field and passes it forward.

2.10. Output Types

OctoSQL makes multiple output formats available, with examples seen in Figures 2.3, 2.4 and 2.5:

- live-table - output in a table format which gets updated as new records and retractions arrive.
- live-csv - output in the CSV format which gets updated as new records and retractions arrive.
- batch-table - same as live-table, but prints only the final state.
- batch-csv - same as live-csv, but prints only the final state.
- stream-json - prints all records as they come, including retractions.

```
SELECT c.description, COUNT(*)
FROM cats_small c
GROUP BY c.description
TRIGGER COUNTING 1
```

Figure 2.3: Example query.

```
+-------------+---------------+
| *star*_count | c.description |
+-------------+---------------+
|           1 | 'fat'         |
|           1 | 'fluffy'      |
|           1 | 'lovely'      |
|           1 | 'the best'    |
|           2 | 'amazing'     |
|           2 | 'cute'        |
|           2 | 'work of art' |
+-------------+---------------+
watermark: 9999-01-01T00:00:00Z
```

Figure 2.4: Example batch-table query output, with the sys.id column omitted.

```
{"*star*_count":1,"c.description":"fat"}
{"*star*_count":1,"c.description":"amazing"}
{"*star*_count":1,"c.description":"the best"}
{"*star*_count":1,"c.description":"fluffy"}
{"*star*_count":1,"c.description":"work of art"}
{"*star*_count":1,"c.description":"lovely"}
{"*star*_count":1,"c.description":"work of art","sys.undo":true}
{"*star*_count":1,"c.description":"cute"}
{"*star*_count":1,"c.description":"amazing","sys.undo":true}
{"*star*_count":2,"c.description":"amazing"}
{"*star*_count":2,"c.description":"work of art"}
{"*star*_count":1,"c.description":"cute","sys.undo":true}
{"*star*_count":2,"c.description":"cute"}
```

Figure 2.5: Example stream-json query output, with the sys.id column omitted.

2.11. Data Sources

2.11.1. New Data Sources

Parquet

Initially, we thought that adding Parquet support would be a small piece of work. In practice, it ended up being very time consuming to understand the ins and outs of the data organization it uses.

Parquet is a column oriented data format encoded with definition and repetition levels based on Dremel[MGL+10]. For readability and performance reasons OctoSQL translates Parquet to objects on its own and relies on an external library only to read Parquet values from the file.

New records are created in the following way:

For each row in a column OctoSQL tries to create a value out of it. If the definition level of the field is too low then the returned value is nil, otherwise it's the value from the Parquet file. If a row contains repeated elements, OctoSQL creates a tuple of values and adds new

ones until repetition level marks the beginning of a new value.

Then a record is created by using row names as variable names and created values as values corresponding to them.

We also support Parquet Logical Types to some extent, as much as it was made possible by the libraries we settled upon initially. This way we understand durations, dates, JSON objects etc. encoded in the Parquet file.

How durability is achieved is described in section 2.11.2.

Kafka

The Kafka datasource implementation is pretty straightforward. For every Kafka partition, an OctoSQL partition is created. When a message is read, a new record is created in the following way:

1. The first field of the record is the message key.
2. The second field of the record is the message offset.
3. Then, the message value is added as a string value (default option) or it can be decoded as JSON and then every field of the decoded object is added (boolean flag $decodeAsJSON$ is needed).

How durability is achieved is described in the 2.11.2 section.

2.11.2. Data Sources Architecture

For many occasions now, we boasted the fact that OctoSQL is an On-and-Off Durable system. But apart from restarting the execution plan using *Stream IDs* described in 2.8, all data sources need their own architecture.

As an example, say we query against a CSV file that contains a 1000 rows and after having processed 500 of them, our machine went off. During the restart of OctoSQL, we run the same query against the same CSV file which results in reading the first half of the file twice, which violates Exactly-Once semantics! A more realistic situation may happen when querying against a SQL database to which we are connected through a network - we all know how tricky network connections can be.

This resulted in using disk storage for all data source states for all supported data sources. In short, it is accomplished with an asynchronous worker, which starts when a *RecordStream* is created and stops when the RecordStream is closed. The RecordStream gets a *batchSize* parameter, which indicates how many records the worker should read from the data source in one storage transaction (by default: 1000). After a successful batch read, the worker moves the special *offset* parameter by the number of records read (this could be less than *batchSize* if he got an *EndOfStream* in the meantime).

To explain how the worker operates, let's consider this simplified code snippet:

```
/* error handling and arguments omitted for simplicity */
func (rs *DataSourceRecordStream) RunWorker() {
    for {
        /* Loading previously saved offset                    */
        offset := rs.loadOffset()
```

```
        /* Loading file or database                                    */
        db := OpenDB()

        /* Moving iterator by offset                                   */
        for i := 0; i < offset; i++ {
            _ := db.getOneRecordAndDoNothingWithIt()
        }

        for {
            /* Read the new records in batch of size rs.batchSize     */
            /* Save the offset moved by currentBatchSize              */
        }
    }
}
```

Now let's notice that loading the offset and moving the "datasource iterator" by its value allows us to skip all the records that have been processed before the crash. After those steps, the worker continues reading records in a normal way. If the worker encounter an error, we simply break the inner for loop and restart the whole worker process.

This schema applies to every iterable data source, which are CSV, JSON, Excel and Parquet files. The "iterating process" is slightly different in SQL, Redis and Kafka data sources.

- Redis worker - when iterating through the whole database, apart from *offset* the worker also saves the current *cursor* (the result of the last scan) which is then used as a next scan parameter to set the database iterator in the right place. But when iterating through some set of keys (where the keys slice is stored by the datasource's RecordStream) it uses $rs.keys[offset:]$ instead of $rs.keys$.
- SQL worker - every SQL query uses $OFFSET$ with a placeholder, which is then substituted by the worker with the loaded offset. So in short, the "iterating process" is skipped here as it is performed by the SQL database when given the offset value.
- Kafka worker - when iterating through a partition, the worker saves the current offset after reading each batch, which can then be used when restarting the Kafka consumer.

Chapter 3

Comparison with Other Systems

In this chapter we provide short comparisons with other potentially similar systems.

The biggest difference is that even though OctoSQL aims to be distributed in the future, it isn't yet. All "competing" projects are distributed. However, they also require much more setup to run than OctoSQL, which is very quick to get started with locally.

3.1. Apache Spark

Described in [ZCF+10], Apache Spark is oriented around in-memory processing, whereas OctoSQL processes everything on-disk. It's also mainly programmable using full-blown programming languages and Spark SQL is meant to ease data processing in such programs. OctoSQL uses SQL for the whole query.

Apache Spark is also orders of magnitude faster than OctoSQL.

3.2. Apache Flink

Described in [CKE+15], Apache Flink uses checkpointing to achieve Exactly-Once guarantees, which become a bottleneck when doing processing with very large amounts of state. This is one of the problems we aimed to tackle with the model of trusting local storage.

3.3. Apache Drill

Described in [HN13], Apache Drill is mainly oriented around batch processing and doesn't provide as much stream-oriented functionality as OctoSQL does.

Apache Drill is also orders of magnitude faster than OctoSQL.

3.4. PrestoSQL

Described in [STS+19], PrestoSQL is mainly oriented around batch processing and doesn't provide as much stream-oriented functionality as OctoSQL does.

PrestoSQL is also orders of magnitude faster than OctoSQL.

3.5. Apache Hive

Described in [TSJ+10], Hive is oriented around batch processing and doesn't provide the stream-oriented functionality OctoSQL does.

Apache Hive is also orders of magnitude faster than OctoSQL.

Chapter 4

Temporal Features

4.1. SQL Dialect

In this section we present special constructions in the SQL dialect used to handle event times.

4.1.1. Triggers

The most basic constructions in our dialect are Triggers (described in detail in section 2.3). They are identified by appearing after the TRIGGER keyword. Here is the list of all the Triggers and their keywords needed to parse them correctly:

- Counting - TRIGGER COUNTING n, where n is a natural number representing the Triggers parameter
- Delay - TRIGGER AFTER DELAY d, where d is a time duration representing the Triggers parameter
- Watermark - TRIGGER ON WATERMARK
- Multitrigger - constructions above combined with a comma, i.e.
  TRIGGER COUNTING n, AFTER DELAY d, ON WATERMARK

4.1.2. Table Valued Functions

Table Valued Functions (in short *tvf*) are a very important dialect construction as they are responsible for creating Watermark Generators (2.5) and Tumbles (2.9.8).

Their parameters are specified using the (param1 => value1, param2 => value2, ...) syntax. Every underlying source of name s is described as $TABLE(s)$, field name t as $DESCRIPTOR(t)$ and all other parameters by the right arrow syntax without keywords. Let's have a look at how to define a tumbling window and both Watermark Generators.

Tumble

- tumble($source$ => TABLE(s), $time\_field$ => DESCRIPTOR(e),
  $window\_length$ => d1, $offset$ => d2)

Where s is the underlying data source and e and $d1$ and $d2$ are Tumble parameters described in the section about Tumble.

Maximal Difference Watermark Generator

- max_diff_watermark(*source* => TABLE(s), *time_field* => DESCRIPTOR(e), *offset* => d)

Where s is the underlying data source and e and d are MDWG parameters described in the section about Watermarks.

Percentile Watermark Generator

- percentile_watermark(*source* => TABLE(s), *time_field* => DESCRIPTOR(e), *events* => n1, *percentile* => p, *frequency* => n2)

Where s is the underlying data source and e and $n1$ and p and $n2$ are PWG parameters described in the section about Watermarks.

4.2. Usage example

Now that we know how to define Triggers and Table Valued Functions we can create a query operating on event times.

Say we have an *events* data source containing information about when a certain team scored a goal. The query below shows how can we group the data by the event time.

First we create a Watermark Source so that we can trigger the data on Watermark. Then, thanks to the Tumble, we can assign time windows by which we will group the records. Next, we can count the goals scored by every team in every time window and trigger the data when we are "almost" sure that all the records that matter were already present ("almost" is never "definitely" as described in the section about Watermarks). In the end, for every time window we want to order by the goals scored, so first we order by the time windows, to maintain event time ordering, and then by the goals.

```
WITH
    with_watermark AS (SELECT *
                       FROM max_diff_watermark(
                                source => TABLE(events),
                                time_field => DESCRIPTOR(time),
                                offset => INTERVAL 5 SECONDS)
                           e),
    with_tumble AS (SELECT *
                    FROM tumble(
                             source => TABLE(with_watermark),
                             time_field => DESCRIPTOR(e.time),
                             window_length => INTERVAL 1 MINUTE,
                             offset => INTERVAL 0 SECONDS)
                        e),
    counts_per_team AS (SELECT e.window_end, e.team, COUNT(*) as goals
                        FROM with_tumble e
                        GROUP BY e.window_end, e.team
                        TRIGGER ON WATERMARK)
SELECT *
FROM counts_per_team cpt
ORDER BY cpt.window_end ASC, cpt.goals DESC
```

Chapter 5

Possible Improvements and Lessons Learned

Building OctoSQL has taken a lot of time and in the meantime we've learned a lot - either by reading or by discovering shortcomings of OctoSQL ourselves. In this chapter we list tradeoffs gone wrong - decisions that we would have made differently knowing what we know now. Multiple of those realisations are based on *How to Architect a Query Compiler, Revisited* [TER18].

5.1. Row Orientation

This is probably the biggest design fault in OctoSQL. It works on streams, which work on a record by record basis in a row oriented fashion. This is conceptually very simple and intuitive but results in many performance shortcomings:

- Virtualisation overhead - nothing is assumed about records, so virtual calls have to be made record by record, field by field. This adds up to a big - non-measured however - performance hit.
- Disk IO - whenever one part of the pipeline sends data to another one, it creates an on-disk record queue, which is filled with records mostly one by one and read the same way. We try to reduce the cost of this by batching processing of multiple records into one transaction, committed atomically, but it's still a high cost to bear and reads are still mostly separate.

These problems could be solved using a column-oriented record-batch format, with batches of records processed simultaneously. However, partly because of the overall design, and partly because of Go as a programming language, this would require rewriting almost all of the components and their corresponding tests.

If we were building OctoSQL anew today, we'd base it on Apache Arrow.

5.2. Serialization Overhead

In Figure 5.1 is the performance profiling output of a sample high-rate workload. We can see that most of the flamegraph is occupied by calls to protobuf serialization/deserialization functions.

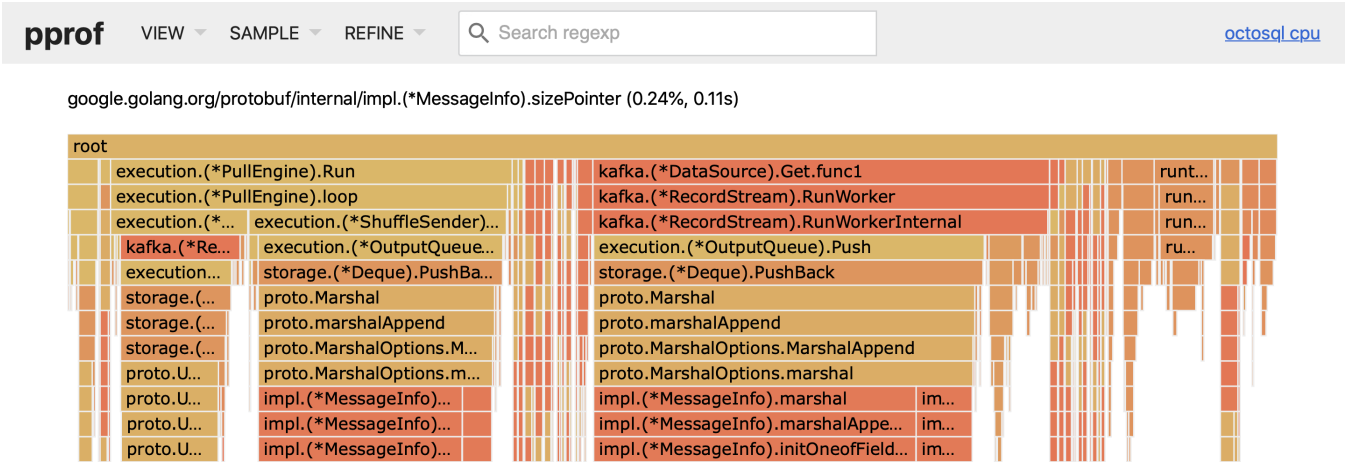


Figure 5.1: Example OctoSQL *go tool pprof* output flamegraph.

Using a graph profile representation in Figure 5.2 we see that a whopping 71% of the time is spent serializing data.

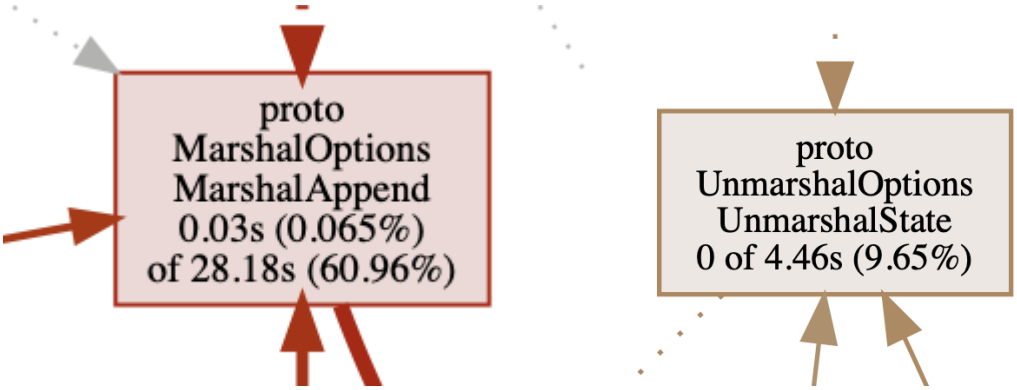


Figure 5.2: Example OctoSQL *go tool pprof* output graph Nodes.

This could be solved by using a serialization-free shared in-memory and on-disk data representation. Again, Apache Arrow comes to mind.

5.3. Lack of Schema

In the beginning OctoSQL took pride in being very dynamic, which made it very flexible. In practice however, queries on data sources with a column that can have varying types usually don't make much sense, at least in a way other than treating it as a string column.

This caused a lot of reflection to be necessary to repeatedly detect the type of a field in a record. This is overhead which could be wholly omitted if schema discovery and schema-based optimisation had been used.

5.4. Non-Composing Storage

In section 2.7 we described how OctoSQL stores all its processing state. We use data structure abstractions which allow us to easily work with serializable messages. This unfortunately

means that those data structures don't compose.

As a result, when we needed a Map with Multisets as values, we had to implement it specifically.

A much better design would be for those data structures to contain other data structures as values, and finally a ValueState could be used as the leaf data structure and store a serializable message.

Chapter 6

Benchmarks

In this chapter we provide some benchmarks of multi-thread scaling. The multithreading is set using the GOMAXPROCS environment variable, which also directly influences Group By parallelism.

All benchmarks have been run using hyperfine [1] on a mid-2019 MacBook Pro 13-inch i7/16GB/256GB which has 4 physical cores with hyperthreading.

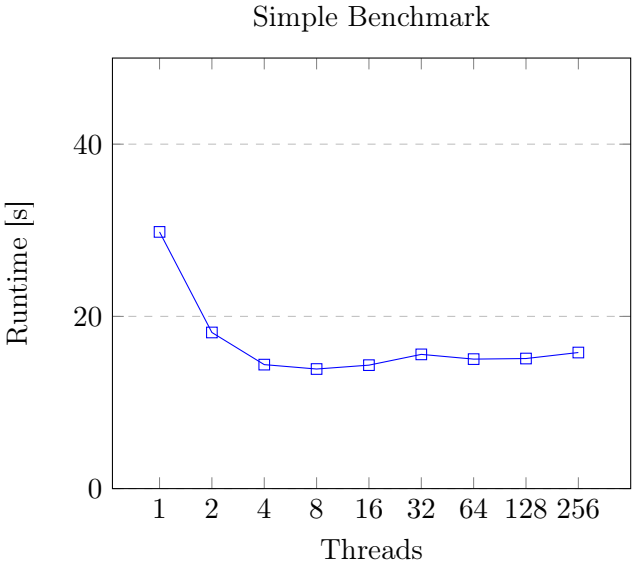
6.1. Simple Query

Here we present a benchmark of the following query:

```
WITH
   with_tumble AS
      (SELECT *
       FROM tumble(source=>TABLE(events),
                   time_field=>DESCRIPTOR(time),
                   window_length=> INTERVAL 1 MINUTE,
                   offset => INTERVAL 0 SECONDS) e),
SELECT e.window_end, e.team, COUNT(*) as goals
FROM with_tumble e
GROUP BY e.window_end, e.team
```

The events table contains 300000 events, which represent goals spread among a 1000 teams and a timespan of a few minutes. The query groups and counts the goals by team and within minute-long windows with the result printed in the end.

[1] https://github.com/sharkdp/hyperfine

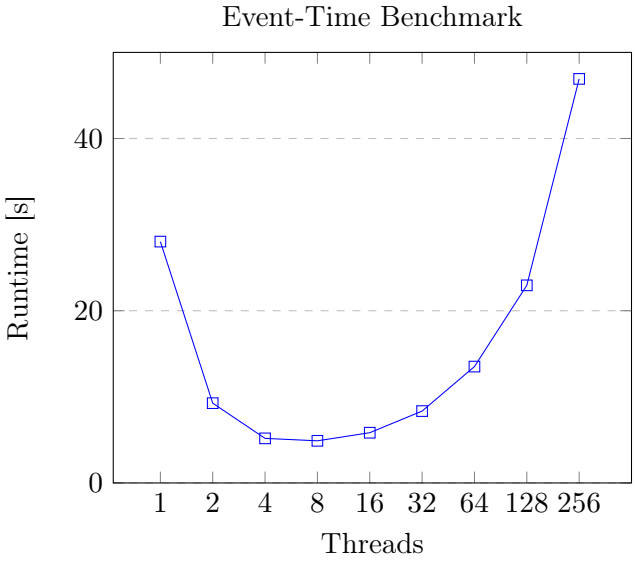


6.2. Query with Event-Time

Here we present a benchmark of the following query:

```
WITH
   with_watermark AS
     (SELECT *
      FROM max_diff_watermark(source=>TABLE(events),
                              offset=>INTERVAL 5 SECONDS,
                              time_field=>DESCRIPTOR(time)) e),
   with_tumble AS
     (SELECT *
      FROM tumble(source=>TABLE(with_watermark),
                  time_field=>DESCRIPTOR(e.time),
                  window_length=> INTERVAL 1 MINUTE,
                  offset => INTERVAL 0 SECONDS) e),
SELECT e.window_end, e.team, COUNT(*) as goals
FROM with_tumble e
GROUP BY e.window_end, e.team
TRIGGER ON WATERMARK
```

The events table contains 25000 events, which represent goals spread among 500 teams and a timespan of a few minutes. The query groups and counts the goals by team and within minute-long windows with the result for a window printed whenever the Watermark for the end of the window arrives.



As we can see, the threading overhead starts to really hurt here if the thread count greatly exceeds the core count.

Chapter 7

Individual Contributions to Octosql

The rules of our bachelor thesis require us to list our specific contributions. Therefore in this chapter you can familiarize yourself with such a list.

7.1. Jan Chomiak

- First Storage Containers (ValueState, Map, Set, List)
- Second Storage Containers (Multiset, Deque)
- Unification of SQL Datasource adding and handling
- Star Expressions
- Distinct
- Stream Join
- Adding IDs to Group By
- Multiple input sources for Pull Engine
- Bugfixes and minor improvements

7.2. Wojciech Kuźmiński

- First Storage Containers (ValueState, Map, Set, List)
- Aggregates
- Max-Diff Watermark Generator
- Percentile Watermark Generator
- Data Sources Durability Architecture
- Close Cleanup
- README Update
- Documentation Update
- Code cleanup and minor improvements

7.3. Rafał Łyżwa

- Parquet Datasource
- Order By

7.4. Jakub Martin

- Pull Engine
- Smart Batching
- Kafka datasource
- Parquet Logical Types addition
- Process By Key
- Lookup Join
- Output Sinks
- Graph Printing Describe
- Triggers: Watermark, Delay, Counting, Multi
- Trigger SQL Extension
- Shuffle
- New Limit and Offset
- New Subquery handling
- Table Valued Functions SQL Extension
- Tumble Table Valued Function
- Common Table Expressions
- Restart Durability using StreamIDs

Bibliography

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. 2015.

[ACL18] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming systems: the what, where, when, and how of large-scale data processing.* " O'Reilly Media, Inc.", 2018.

[AGZ+15] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph Bradley, Xiangrui Meng, Tomer Kaftan, and Michael Franklin. Spark sql. pages 1383–1394, 05 2015.

[BAH+19] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. One sql to rule them all - an efficient and syntactically idiomatic approach to management of streams and tables. *Proceedings of the 2019 International Conference on Management of Data - SIGMOD '19*, 2019.

[CEF+17] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[Gra94] Goetz Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[HN13] Michael Hausenblas and Jacques Nadeau. Apache drill: Interactive ad-hoc analysis at scale. *Big Data*, 1:100–104, 06 2013.

[HSW13] Jr. Henry S. Warren. *Hacker's Delight Second Edition.* " Pearson Education, Inc.", 2013.

[LPG+17] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.

[SKRC10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[STS+19] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. Presto: Sql on everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1802–1813, 2019.

[TER18] Ruby Y Tahboub, Gregory M Essertel, and Tiark Rompf. How to architect a query compiler, revisited. 2018.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.