ILLINOIS TECH

POLITÉCNICA

ETS INGENIEROS INFORMÁTICOS

ChronoSQL: A SQL interpreter for the ChronoLog project

GitHub

Pablo Pérez Rodríguez

08/11/2022

Index

1. Introduction

Streaming SQL [1][2]

- Growing need to manage high volumes of data
- Streaming platforms (Kafka, Kinesis) have emerged to provide a solution
  - Operations are done through their own APIs
  - + Powerful computation capabilities
  - - Steeper learning curve
- SQL offers a way to query streaming data
  - Standardized programming language
  - One of the most popular languages[3]
  - Can be adopted more easily by people from different backgrounds[3]
- Many platforms have been developed to offer this functionality recently
  - ksqlDB
  - Materialize
  - …

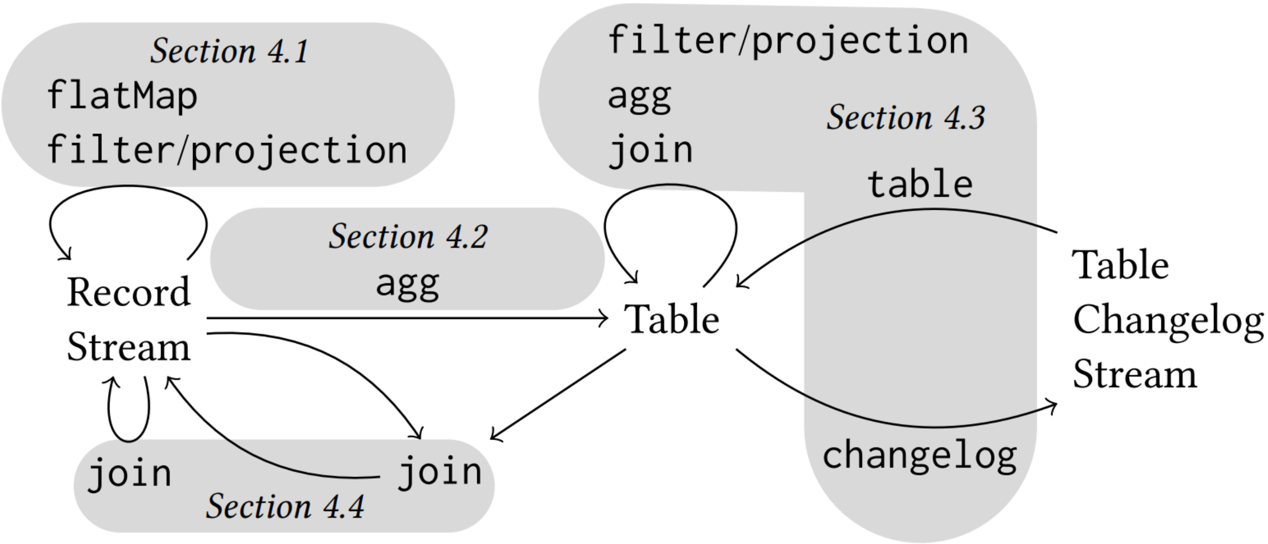
Initial goals

- Design and implement a library to fetch data from ChronoLog using SQL
    - Read and parse SQL statements
    - Interpret the query tree
    - Execute the ChronoLog operations to fetch the corresponding data
- Focused on time
    - Time series data
    - ChronoLog’s physical time design

2. State of the art

Streaming SQL key concepts

- Stream-table duality [4]
  - Stream: append-only series of facts (events)
  - Table: static view of the state of a stream at a certain point in time



Streaming SQL key concepts

- Materialized views
  - Very common in most engines studied - similar concept as in traditional DBs
  - They are continuously updated on the background to provide low-latency up-to-date results
    - Usually automatic and incremental, opposed to the traditional interval model
  - Very efficient
    - Updates are handled as events arrive incrementally
    - Queries fetch already computed data
- Relational vs streaming queries
  - Relational query
    - Returns the current state of a stream/table and terminates
  - Streaming query
    - Subscription to real-time changes on a stream
    - Changes are emitted as they arrive (until the process is terminated)
    - Useful, for example, to feed data to a dashboard

Existing solutions

- [ksqlDB]
  - Developed by Confluent (original developers of Apache Kafka)
  - SQL over a Kafka cluster
- [Materialize]
  - SQL over Kafka, Kinesis (in preview), or local files
  - As its names suggests, centered around materialized views
    - Differential dataflow model
- [SamzaSQL]
  - SQL on top of Apache Samza (a distributed stream processing framework)
  - Can be integrated with multiple sources (custom ones as well)
- Presto
  - Distributed SQL query engine
  - Can be integrated with most RDBMS, stream platforms, Hadoop…

Existing solutions comparison

Features	ksqlDB	Materialize	SamzaSQL	Presto
Relational queries	Y	Y	Y	Y
Streaming queries	Y	Y	Y	N
Queries on streams	Y	Y	N	Y
Materialized views	Y	Y	N	N
Cluster mode	Y	N	N	Y
Window functions	Y	N	Y	Y

3. Design

Assumptions

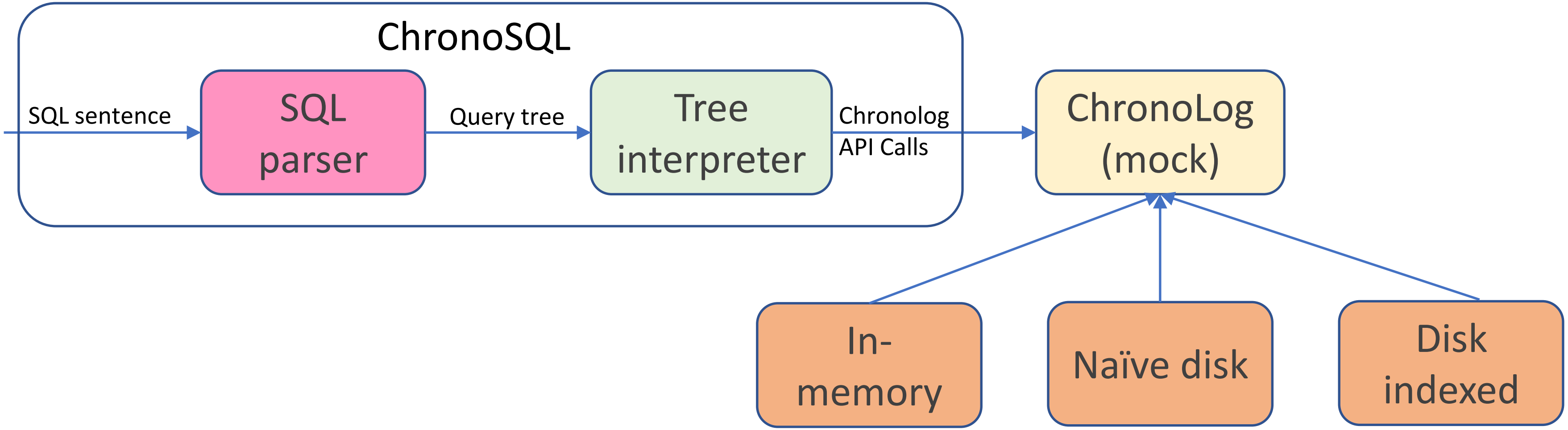
- Event data is uninterpreted
    - We know nothing about what the payload bytes represent
- Queries will only use the ChronoTick (timestamp) of events
- ChronoTick is represented as a std::time_t
    - Integral value
    - Usually used to hold number of seconds since 00:00, Jan 1, 1970 (Unix time)
    - Implemented as a long int
    - Granularity of seconds
- Event data will have a fixed size
    - Simplicity purposes
    - Non-fixed sizes could be implemented relatively easily in the future

Initial goals

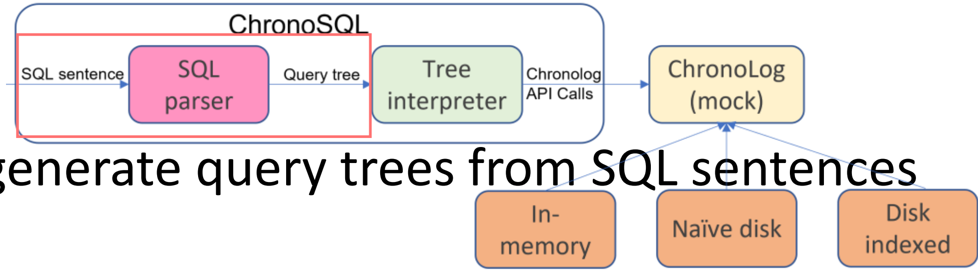
- Design and implementation were focused on building a system to support the following 5 initial queries:
  1. GetAll
     - Simple selection
     - SELECT * FROM <log_name>;
  2. GetByTime
     - Time filtering
     - SELECT * FROM <log_name> WHERE EID (>, <, >=, <=) <timestamp>
  3. CountAll
     - Aggregation
     - SELECT COUNT(*) FROM <log_name>;
  4. GetWeekend
     - Day of the week filtering
     - SELECT * FROM <log_name> WHERE EID = ‘Saturday’ OR EID = ‘Sunday’;
  5. CountByDay
     - Windowing (grouping by day) and counting
     - SELECT window('1 day’), count(*) AS one_day FROM <log_name> GROUP BY one_day;

High-level design

- Three main sub-components
  - SQL parsing library
  - Tree interpreter
  - Chronolog (mocked with three implementations)

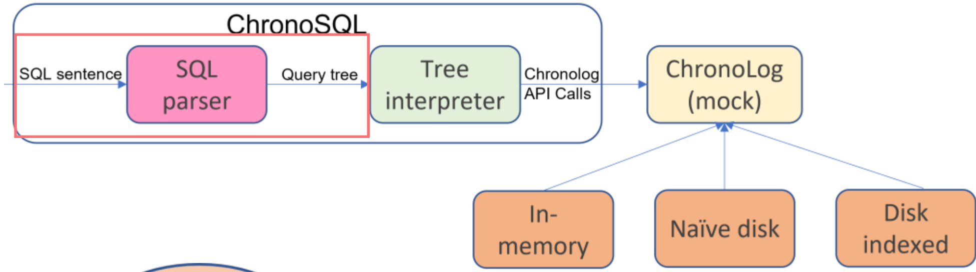


Parsing SQL statements

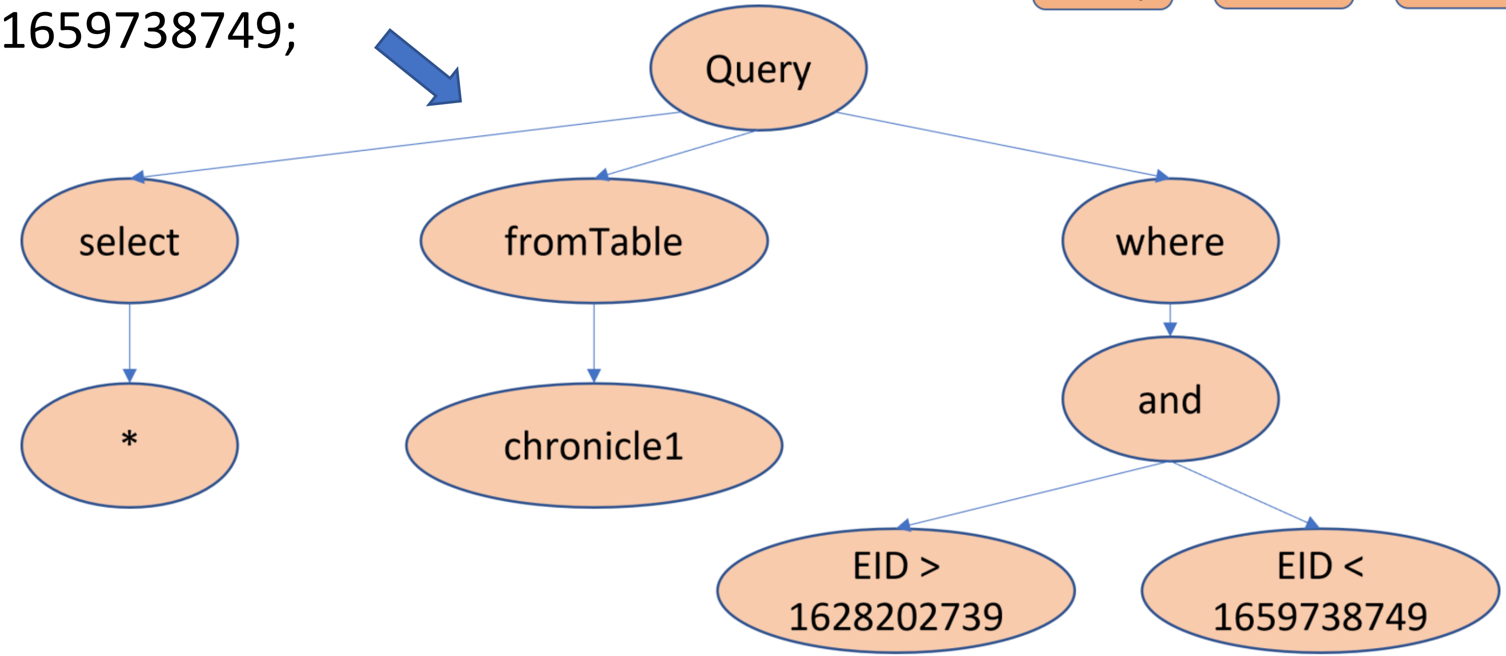


- It is used to parse, validate and generate query trees from SQL sentences
- Use of an external library
  - Less development time
  - No need to reinvent the wheel
- Several libraries were explored and compared
  - sqltoast
  - usql
  - Hyrise SQL parser
  - libpg_query
- The Hyrise SQL parser was chosen because:
  - Easy to include new functionality
  - It keeps being periodically updated
  - Wide SQL support
  - Small library size

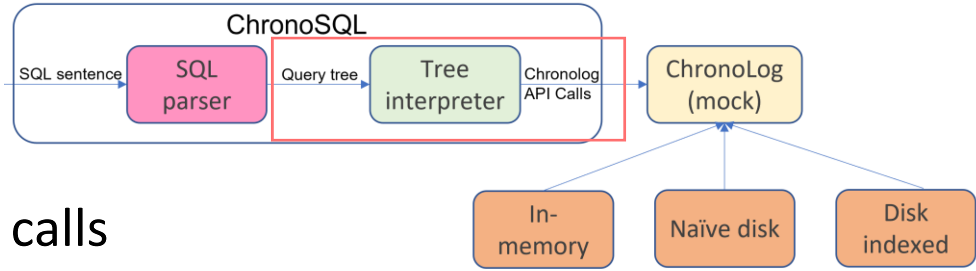
Parsing SQL statements



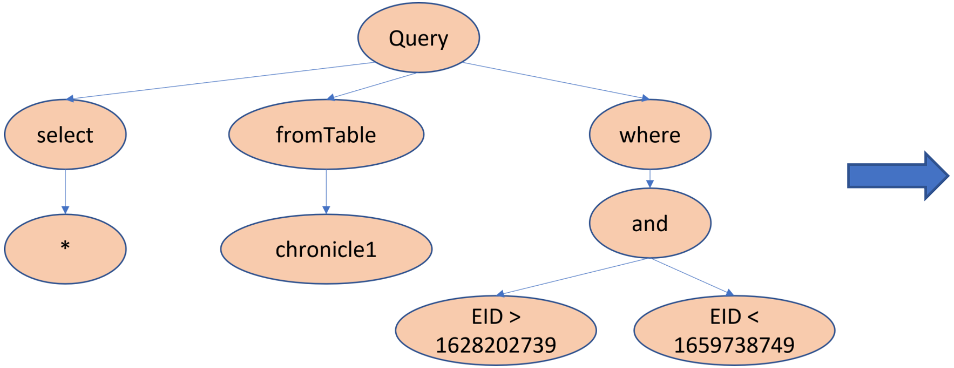
SELECT * FROM chronicle1
WHERE EID > 1628202739
AND EID < 1659738749;



ChronoSQL

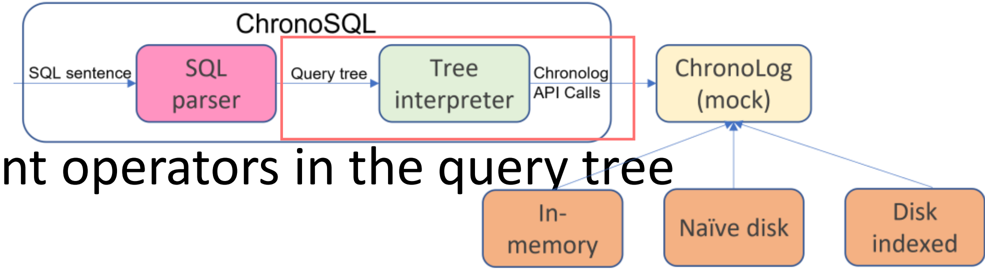


- Interprets the query tree
- Transforms it into Chronolog API calls



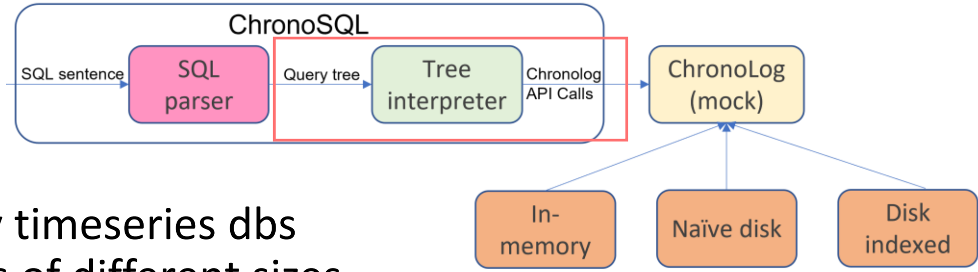
Chronolog.replay("chronicle1", {1628202739, 1659738749})

ChronoSQL



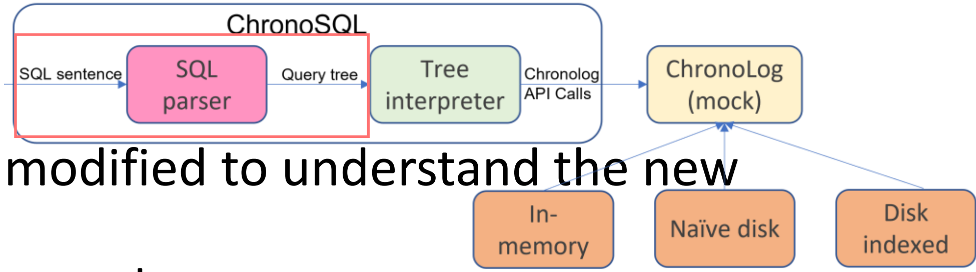
- ChronoSQL identifies the different operators in the query tree
  - Selection
  - Filtering
  - Grouping
  - Functions
- Some of this information is used to fetch data from ChronoLog
  - ChronoLog only receives a chronicle and, optionally, min and/or max EIDs
- The rest of the data processing is done by ChronoSQL
  - Counting events (aggregation)
  - Windowing events (grouping)
  - Day of the week filtering

ChronoSQL – time



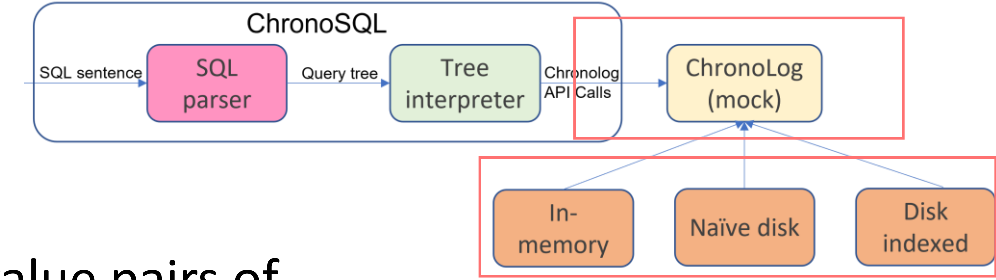
- Windowing
  - Functionality supported by many timeseries dbs
  - Events are grouped into windows of different sizes
    - Second, minute, hour, day, month, year
  - Applied after the events are fetched from ChronoLog
- Days of the week
  - ChronoSQL recognizes days of the week
  - The day of the week can be extracted from the EID
    1. Transform from seconds since 1970 to days since 1970 -> day = floor(EID / 86400)
    2. Add 4, because January 1st, 1970, was Thursday -> day = day + 4
    3. Apply mod 7 to get the day of the week -> week_day = day % 7
  - Useful to, for example, "Find the events of every Monday in the last month"

Extending the parser



- The SQL parser library had to be modified to understand the new operators
- The Hyrise parser uses Bison, a general-purpose parser generator
- First, the Bison grammar needed to be changed
  - Including the new syntax rules
- Then, the new expressions had to be included in the .h files
  - Interval expression (1 day, 2 months, etc.)
  - Days of the week (enum)
- Finally, modify the API to return the new fields

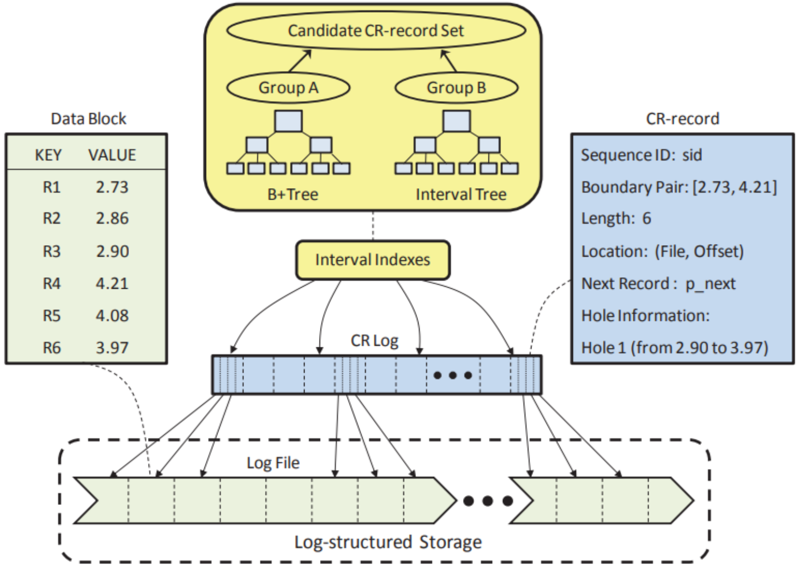
Data storage



- ChronoLog mock
  - Simulate retrieval of data
- Events are represented as key-value pairs of
  - ChronoTick, std::time
  - Payload, const char* (uninterpreted)
- Initially, we created two implementations of the ChronoLog API:
  - In-memory storage
  - Naïve disk storage
- Then, we tried to speed up the disk storage using indexing

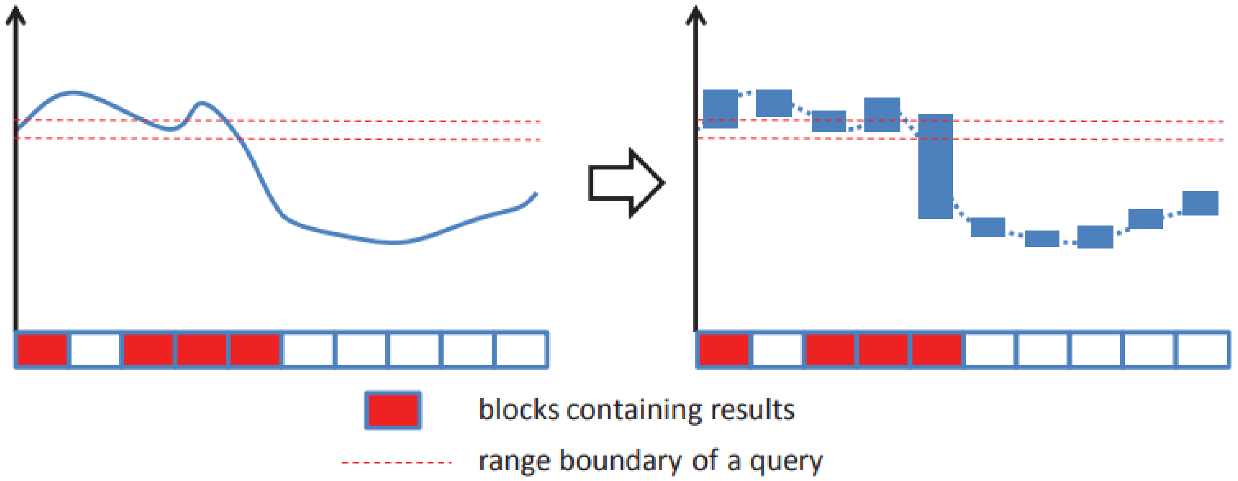
Indexing timestamped events [5]

- Continuous Range Index (CR-index) – Introduced by [5]
- The log is (logically) split into groups of records
- One index entry per group
    - Lower and upper boundary
    - Length
    - Offset
- Block lenght can be adapted
- This index can be indexed as well
    - LSM trees
    - B+ trees
- Kafka uses a very similar approach



Indexing timestamped events

- Translation of a query
  - From continuous data to blocks of data
  - A query will be matched against the different blocks of events
  - Each block that has at least one element matching the criteria will be fetched
  - Fetched blocks are examined to find events that meet the criteria

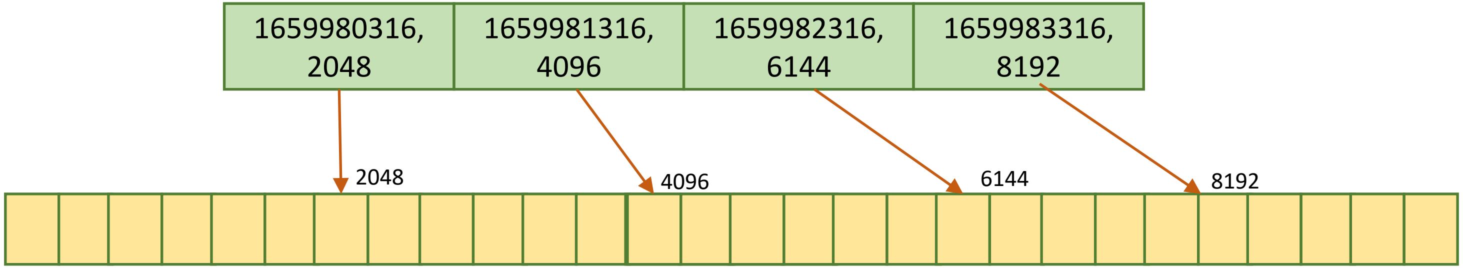


CR-indexing Benefits [4]

- Lightweight
  - Size is 5-10% of an LSM-tree/B+ tree
- Low write overhead
  - No more than 8%
  - Compared to 45-77% LSM, 78-124% B+ tree
- Query response time in line with LSM-tree and B+ tree
  - Lower index-lookup cost (many less entries)
  - Higher data access cost

Indexing in our Chronolog mock

- Events are written to a log file like in the naïve implementation
- Once a byte threshold is reached, an entry is appended to the index
  - Also written to disk to have a backup



- To look for an event (or range of events)
  - Look for the closest timestamp in the index
  - From that offset, search for the actual event (or start of range)

4. Evaluation

Dataset

- Generated randomly
  - Along 365 days (from August 5th, 2021, to August 5th, 2022)
  - Distributed uniformly
- Several sets with different number of events (10k, 100k, 1m, 10m)

Environment

- OS
  - Manjaro (5.10.133-1-MANJARO)
  - Virtual Machine
- Hardware
  - 8 cores, 16 threads
  - 12GB RAM
  - 512GB PCIe 3.0 NVMe M.2 SSD
- Software
  - C++17
  - gcc version 12.1.0

Queries – basic set

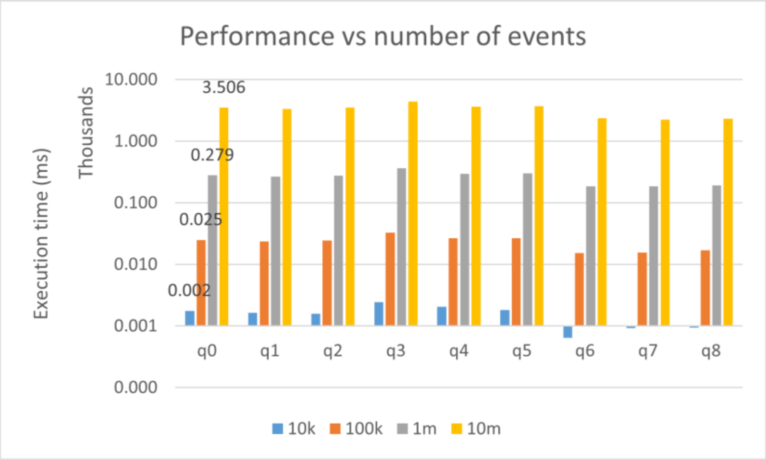
- We have used the 5 initial queries:
  0. GetAll
     - Simple selection
     - SELECT * FROM <log_name>;
  1. GetByTime
     - Time filtering
     - select * from <log> where EID > 1628780806 AND EID < 1658105202;
  2. CountAll
     - Aggregation
     - SELECT COUNT(*) FROM <log_name>;
  3. GetWeekend
     - Day of the week filtering
     - SELECT * FROM <log_name> WHERE EID = 'Saturday' OR EID = 'Sunday';
  4. CountByDay
     - Windowing (grouping by day) and counting
     - SELECT window('1 day'), count(*) AS one_day FROM <log_name> GROUP BY one_day;

Queries – more advanced set

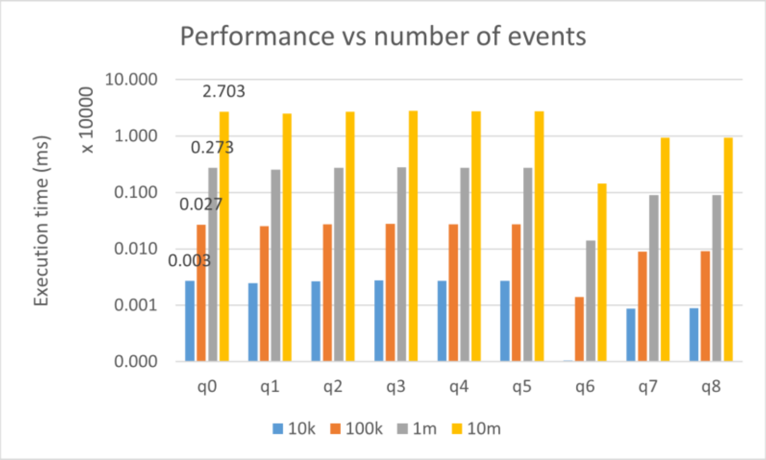
- We also included more advanced queries
    5. CountByMonth
        - Wider windowing (1 month)
        - select window(1 month) as one_month, count(*) from <log_name> group by one_month;
    6. GetByTimeEnd
        - Filtering by an EID towards the end of the chronicle
        - select * from <log> where EID > 1658105202;
    7. CountByDayFiltered
        - Windowing, aggregating and filtering by timestamp
        - select window(1 day) as one_day, count(*) from <log_name> where EID > 1641936187 and EID < 1652395282 group by one_day;
    8. CountMonday
        - Windowing, aggregating and filtering by day of the week
        - select window(1 day) as one_day, count(*) from <log_name> where EID = 'Monday' and EID > 1641936187 and EID < 1652395282;

Results – Increasing number of events

- Linear increase in response time
- The greatest part of the execution time is employed fetching data
- Minimal overhead from SQL operations



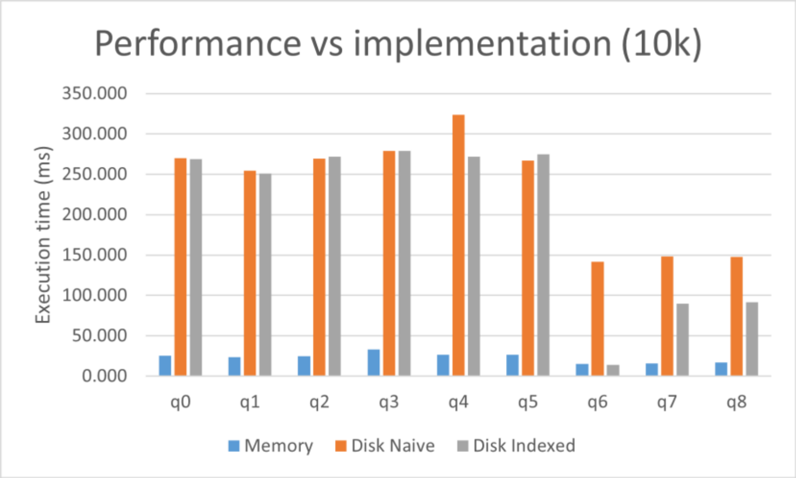
Memory

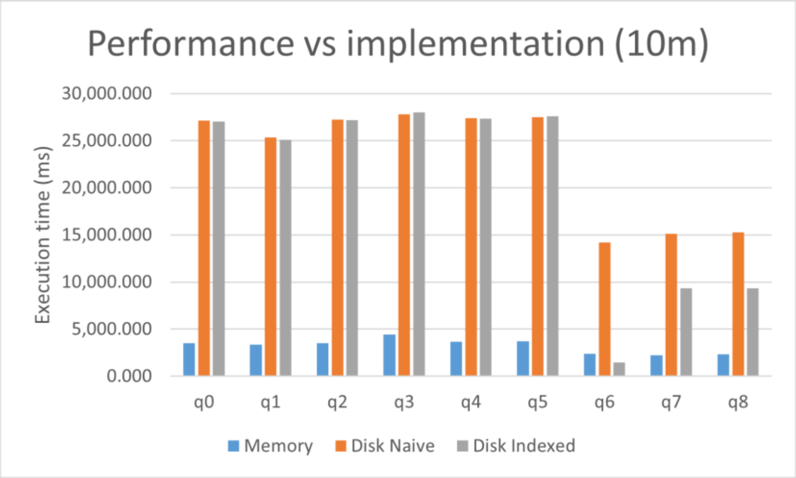


Disk indexed

Results – Implementation comparison

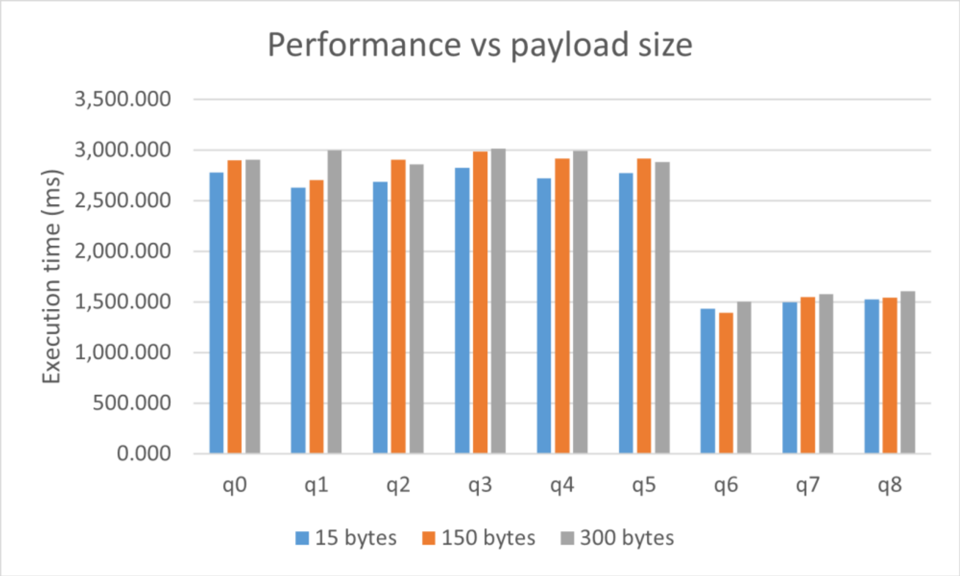
- Memory is significantly faster than disk
- Indexing can reduce execution time to a 10% in certain queries
  - Queries that only require a subset of the entire log
  - For others, it is not noticeable





Results – Modifying the fixed event size

- Larger event sizes do not make measurable differences in execution times



5. Conclusion and future work

Future work

- Materialized views and streaming queries
- Distributed implementation
- Wider SQL syntax support
- Query optimization

6. References

References

1. S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, 10 "Benchmarking streaming computation engines: Storm, flink and spark streaming," in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 1789–1792
2. G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, and J. Stein, "Building a replicated logging system with apache kafka," Proc. VLDB Endow., vol. 8, no. 12, p. 1654–1655, aug 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824063
3. M. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, "Streams and tables: Two sides of the same coin," Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics, 2018.

References

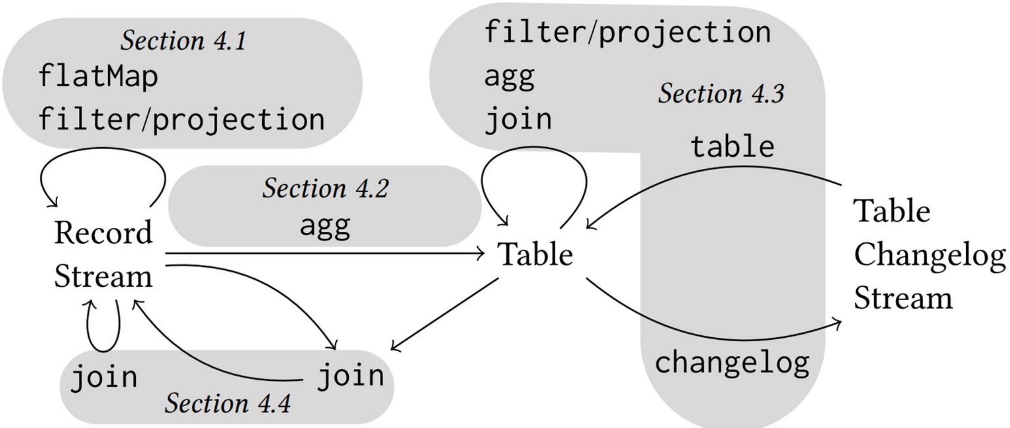
4. Stack Overflow, “Stack Overflow Developer Survey 2021,” 2021. [Online]. Available: https://insights.stackoverflow.com/survey/2021#overeview

5. Sheng Wang, David Maier, and Beng Chin Ooi. 2014. Lightweight indexing of observational data in log-structured storage. Proc. VLDB Endow. 7, 7 (March 2014), 529–540. https://doi.org/10.14778/2732286.2732290

Stream-table duality

- Table
  - Static view on the result of an operator up to an offset, updated as inputs are processed
- Table changelog
  - Dynamic view on the result of an operator, records are updates to a table
- Stream
  - Records represent facts (and not updates)

- Important tradeoff
  - Always working with materialized views
  - Sometimes operating over the table changelog stream only
- Operators may produce transitions (for example, from stream to table) or not



Basic concepts

- Streaming vs relational queries
  - Streaming query
    - Subscription to real-time changes on a stream
    - Changes are emitted as they arrive (until the process is terminated)
  - Relational query
    - Returns the current state of a stream/table and terminates
- Materialized views
  - Very common in most engines studied - similar concept as in traditional DBs
  - They are continuously updated on the background to provide low-latency up-to-date results
    - Usually automatic and incremental, opposed to the traditional interval model

Platforms overview

- ksqlDB
  - Developed by Confluent (original developers of Apache Kafka)
  - SQL over a Kafka cluster
- Materialize
  - SQL over Kafka, Kinesis (in preview), or local files
  - As its names suggests, centered around materialized views
    - Differential dataflow model
  - pgwire (PostgreSQL wire protocol) to connect Postgres CLI to Materialize (psql, for example)
- SamzaSQL
  - SQL on top of Apache Samza (a distributed stream processing framework)
  - Integrable with Kafka, Kinesis, EventHubs, ElasticSearch, Hadoop
    - In theory, it can be integrated with custom sources
- Facebook Presto
  - Distributed SQL query engine
  - Can be integrated with most RDBMS, stream platforms, Hadoop…
- Apache Pinot
  - Real-time distributed OLAP datastore to query data using SQL-like syntax (PQL)
  - Initially developed by LinkedIn to handle internal and user-related queries
  - Lambda architecture

Feature table

Features	ksqlDB	Materialize	SamzaSQL	Presto	Pinot
Streaming queries	Yes	Yes	Yes	No	No
Relational queries	Yes	Yes	Yes	Yes	Yes
Insert/delete statements	Yes	Yes (but data is not persisted)	Yes	Yes	No
Materialized views	Yes	Yes	No	Trino	No
Queries on streams	Yes	No	Yes	Yes	-
Cluster mode	Yes	No (yet)	No	Yes	Yes
External platforms	RocksDB Kafka topics Zookeeper	psql CLI pgwire	Zookeeper Calcite YARN	No	Zookeeper Helix

Feature table

Features	ksqlDB	Materialize	SamzaSQL	Presto	Pinot
Hopping and tumbling windows	Yes	No	Yes*	Yes	No
Sliding windows	Yes	No	Yes*	Yes	No
Stream-to-stream joins	Yes (windowed)	Yes	Yes*	Yes	-
Stream-to-relation joins	Yes	Yes	Yes*	-	-
Relation-to-relation joins	Yes	Yes	No	-	-
User-defined functions	Yes	No	Yes	Yes	No
Pluggable to different sources	No	Yes	Yes	Yes	Yes
Service/library	Service	Service	Library	Service	Service

* = not implemented yet

Interesting optimizations - ksqlDB

- Can be deployed as a cluster having several instances running in parallel
  - No master node needed
  - Cluster can be scaled out even while an operation is being executed
- Uses Kafka topics to store information
  - Creates a topic for each stream and table declared
  - (Compacted) Changelog topics to persist aggregations and similar operations updates
    - Compaction periodically deletes all but the latest record per key
- Also stores state data in RocksDB
  - The current state of a table (materialized) is stored ephemerally to RocksDB
  - If there is a failure, it can recover the latest state of a materialized view applying the compacted changelog topic from the beginning
- Latest-value materialization (LATEST_BY_OFFSET operator)

Interesting optimizations - Materialize

- Differential dataflow
  - Data-parallel programming framework
    - Scales the computation from a single thread to a cluster of nodes
    - Designed to process large volumes of data efficiently
  - Used to update materializations whenever data changes occur
  - Incremental updates - source is polled for updates
- Join optimization
  - Makes heavy use of indexes to optimize joins
  - Almost every primary and foreign key needs to be indexed to make use of their optimizations
    - Initial storage and computation overhead is high, but it is likely amortized over time
  - For a fully optimized query, it only requires to store 2x the number of final records in each materialized view
- Compaction
  - Materialize stores materialized views entirely in memory
  - To prevent over-usage, it compacts data based on compaction windows

Interesting optimizations - SamzaSQL

- Uses Apache Calcite
  - SQL parsing, validation, planning and optimizing
  - Calcite’s output is then translated to Samza Model
  - SamzaSQL added some streaming extensions to it
- SQL shell was built using SqlLine
- In theory, easily pluggable to custom sources
- Uses Zookeeper to store metadata and configuration information
  - This info is then shared between query planner and SamzaSQL streaming tasks

Interesting optimizations - Presto

- Master-slave architecture
  - Master node
    - Parsing, planning and query optimization
    - Coordinates and manages how queries are executed
- Capable of processing data from more than one source simultaneously
- Presto has no built-in fault tolerance
  - These mechanisms consume notable resources
  - Presto is built for very high performance
  - They assume the benefits of fault tolerance management do not make up for the resource consumption

Apache Calcite

- Used by Hive, Drill, Storm, SamzaSQL
- Provides an entire query processing system
  - Query parsing, validation, planning and optimization
- Extendable (rules, planners, operators, etc.)
- Provides streaming functionality
  - Streaming queries
  - Window functions
  - Streaming joins
- Some functionalities, such as certain joins, have not been implemented yet

Interesting optimizations - Time series databases

TimescaleDB

- Partitioning into hypertables and chunks
  - Most recent data can be kept always in memory
    - Inserts and queries to recent data are faster
    - Smaller indexes that can also be kept in memory easily
- Continuous aggregates (real-time aggregation)
  - Materialized views can be refreshed on an established basis
  - Real time aggregation merges materialized aggregates with raw data
    - Accurate and up-to-date results
    - Benefits from pre-computed aggregates for a large portion of the result
- Time ordering optimization
  - As the time range is known for every chunk of data, queries can stop accessing chunks once the needed number of rows is reached (in LIMIT clauses)
  - Sorting can also be optimized because only the data inside a chunk needs to be sorted

Pending issues

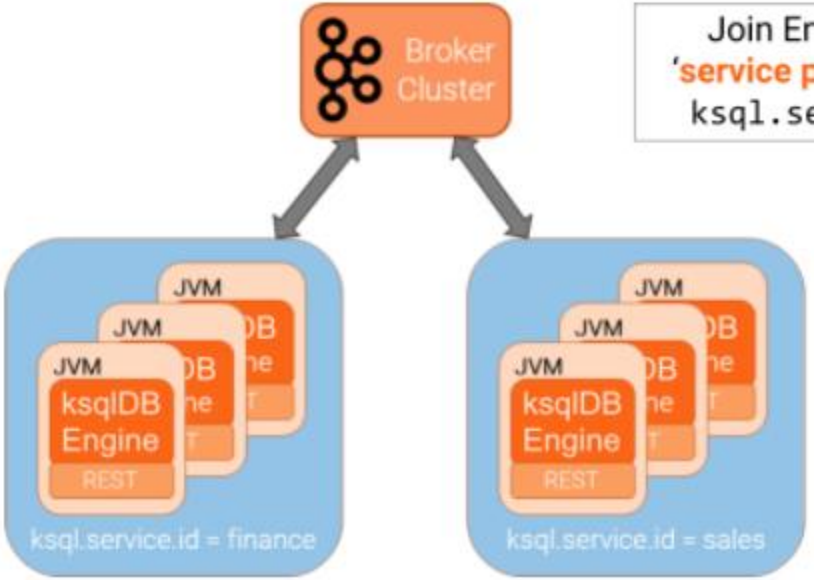
- Push vs pull clients in streaming queries
- Operators and APIs of the systems mentioned
    - Common functionality among platforms (the musts)
- More timeseries databases optimizations (InfluxDB - kdb+)
- Design 5 queries, growing complexity, to drive design and development
    - Start with a single log, then increase the number of them

Background presentation

04/06/2022

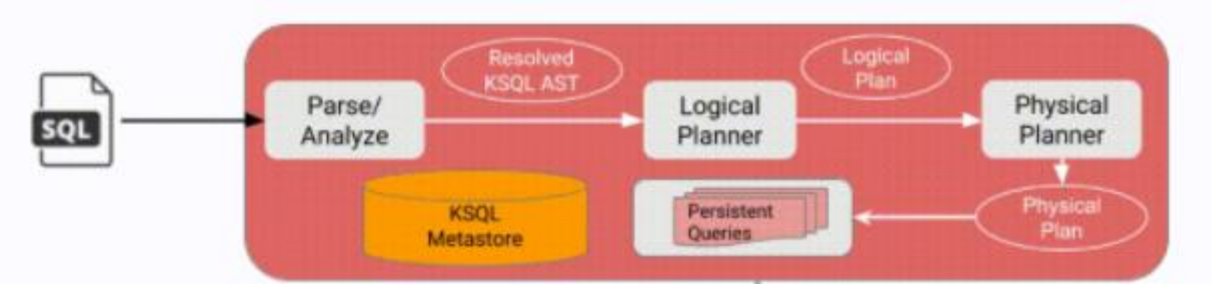
ksqlDB deployment

- ksqlDB allows to have several engines running in parallel
- No master node or coordination is required
- A cluster can be scaled out even while an operation is being executed
- ksql.service.id allows to create multiple ksql clusters connected to the same Kafka cluster
- Basic ksqlDB server general guidelines:
    - 4 cores
    - 32GB RAM
    - 100GB SSD
    - 1 Gbit network



ksqlDB query lifecycle

1. A stream linked to a kafka topic needs to be registered in the first place to query data from
2. SQL statement is executed against an existent stream
3. Engine parses the SQL statement into an abstract syntax tree (AST)
   a. The parser is based on ANTLR
4. Engine creates the logical plan using the AST
5. Engine transforms the logical plan into a physical one
   a. Basically, translates it into a Kafka Streams application



ksqlDB query lifecycle - example

1. Stream is created

```
CREATE STREAM authorization_attempts
  (card_number VARCHAR, attemptTime BIGINT, ...)
  WITH (kafka_topic='authorizations', value_format='JSON');
```

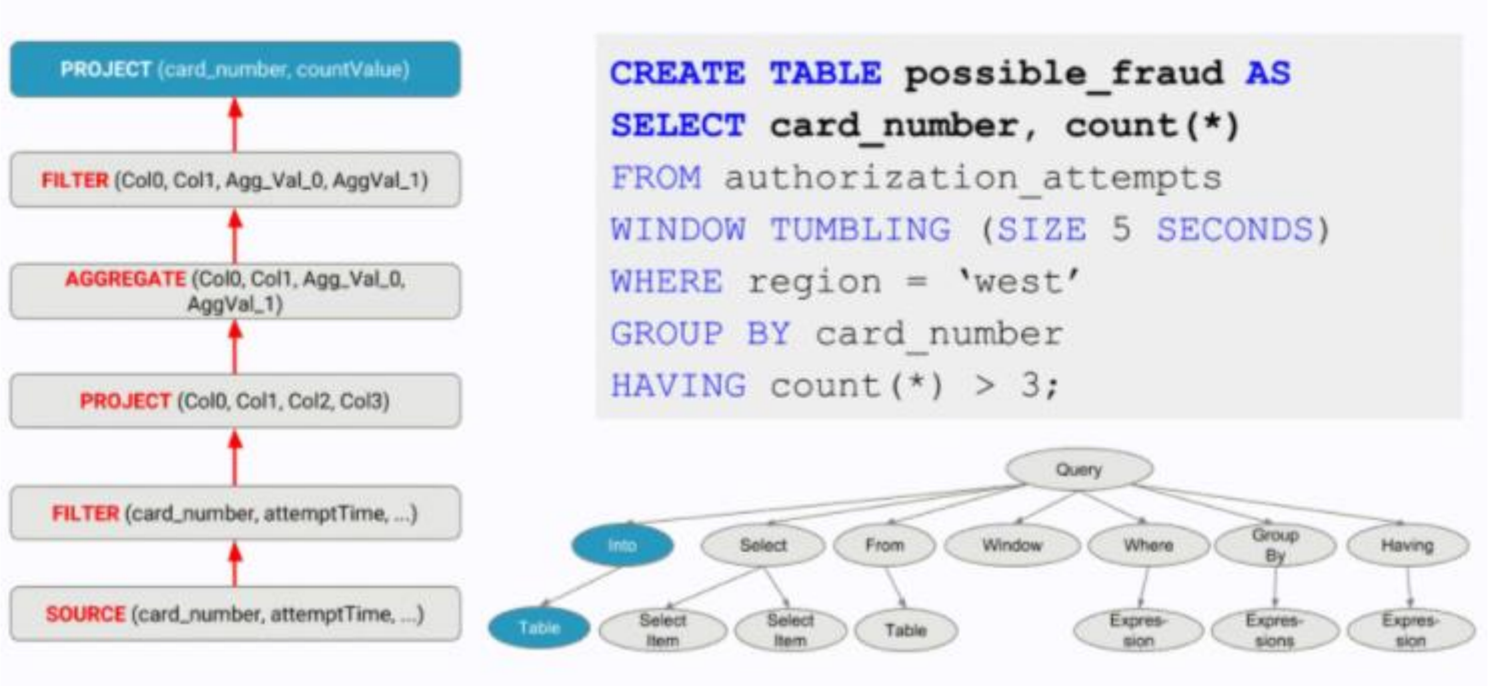
1. SQL statement to fetch data from created stream

```
CREATE TABLE possible_fraud AS
  SELECT card_number, count(*)
  FROM authorization_attempts
  WINDOW TUMBLING (SIZE 5 SECONDS)
  WHERE region = 'west'
  GROUP BY card_number
  HAVING count(*) > 3
  EMIT CHANGES;
```

1. Statement is parsed

ksqlDB query lifecycle - example

4. Logical plan is created



5. Physical plan is created - basically translate logical steps to Kafka Streams operators

ksqlDB data - topics

- Output topic - every created Stream and Table results in the creation of a Kafka topic
  - Same name as stream/table by default
  - 4 partitions by default
  - Replication factor of 1 by default
- Internal topic - some queries require that the input stream be repartitioned so that all messages being aggregated or joined together reside in the same partition
  - Intermediate topic is created and every record is produced to and consumed from that topic
  - It has the same number of partitions and replicas as the input stream/table
- Internal topic - changelog topics. Aggregations and similar operations' state is persisted in a compacted changelog topic.
- Internal topic - commands topic where the log of queries sent in interactive mode is persisted

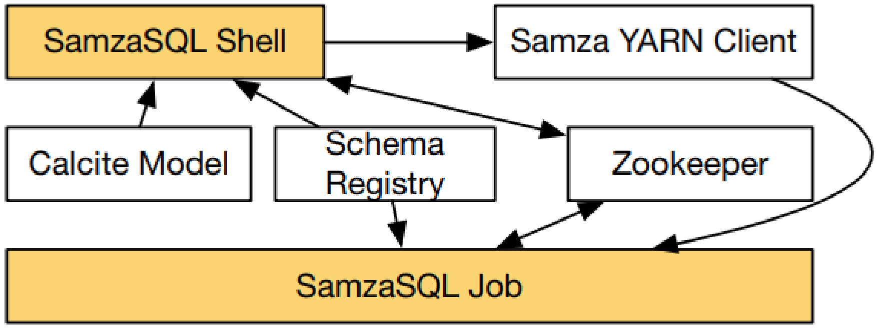
ksqlDB data - RocksDB

- Embeddable key-value store
- Used to store state for computing aggregates and joins
- ksqlDB creates one or more RocksDB instances to store the state for each stateful task in a query

SamzaSQL

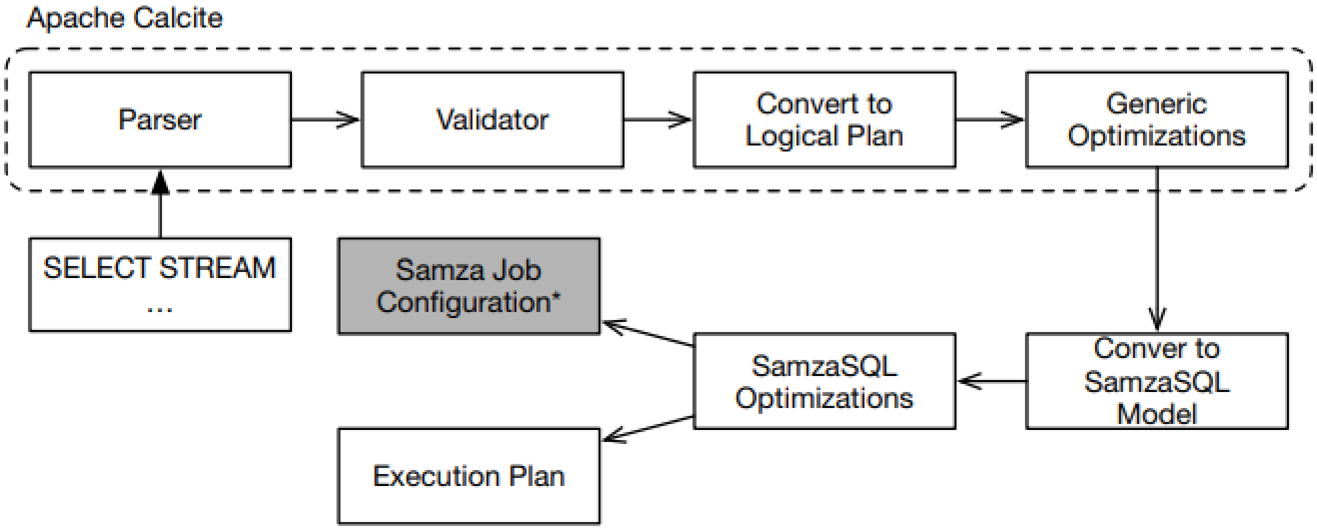
- Scalable SQL streaming query engine
- Both batch and streaming data
- Pluggable to many sources (in theory, easy to integrate with custom sources)
- Originally developed at LinkedIn
- Built on top of Apache Samza
  - Stream processing framework - can be connected to Kafka, among others
- Uses Apache Calcite
  - Added some stream extensions
  - Open source -> can be checked to see how to use Calcite
- Support for collection types
- Also has the same concept of “push” vs “pull” queries
  - Streaming queries - tuples are outputted whenever they arrive (it keeps running)
  - Non-streaming queries - consider the stream as a table with the history of the stream up to the execution of the query

SamzaSQL architecture



- SQL shell
  - Built using SqlLine
- Apache Calcite:
  - Parse query
  - Validate query
  - Create logical plan
  - Apply some generic optimizations
- Optimized plan is converted into a physical plan
  - Convert to SamzaSQL Model
- Some metadata is stored in Zookeeper in the job config. generation step
  - Message schemas, streaming query
  - Tasks then read this configurations from Zookeeper

SamzaSQL architecture

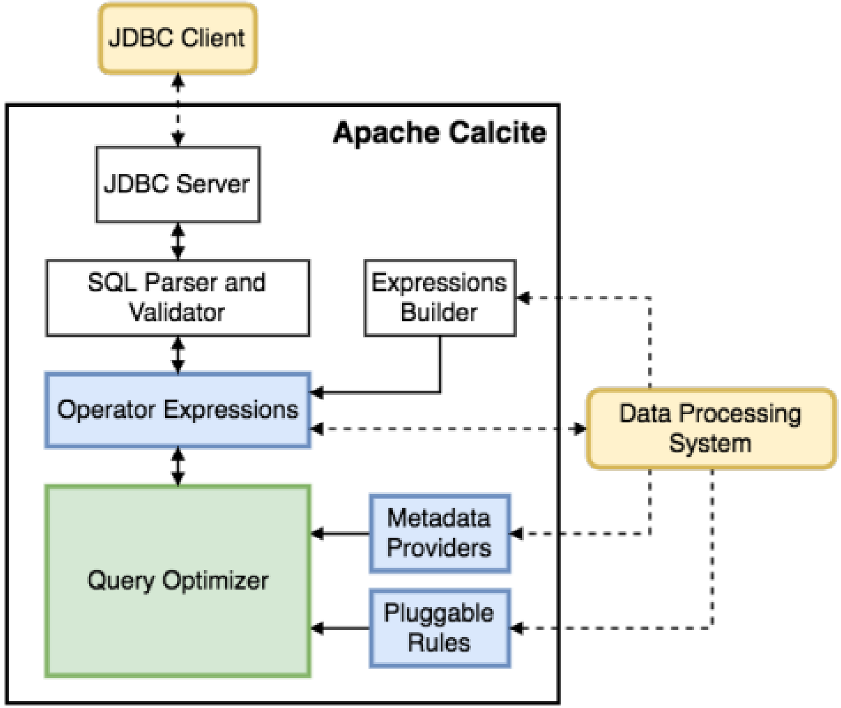


Apache Calcite

- Used by Hive, Drill, Storm, SamzaSQL, HerdDB
- Query processing system
  - Query execution
  - Query optimization (flexible query optimizer, pluggable and extensible)
  - Query languages
- Open source
- Multiple data models supported
  - Conventional (batch) data processing
  - Streaming data processing
- SQL with extensions
- JDBC conformant
- Does not provide a repository for storing metadata

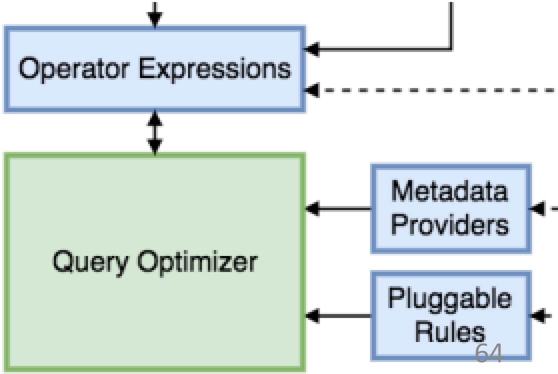
Apache Calcite architecture

- Parser and validator translate SQL to a tree of relational operators.
- Query Optimizer optimizes the tree using planner rules, metadata and planner engines
- Calcite can work in optimize-only mode
  - Once the query has been optimized, Calcite translates the tree back to SQL.



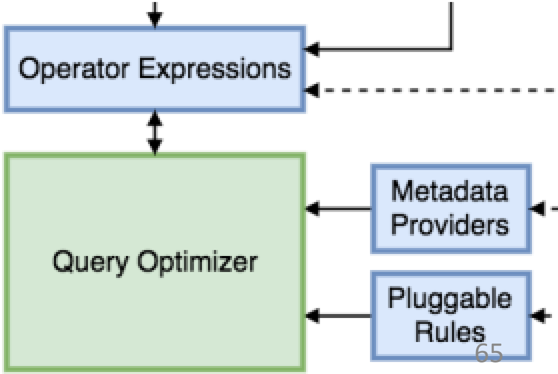
Apache Calcite query optimizer

- Planner rules: queries are optimized applying planner rules repeatedly
  - A rule matches a pattern in the tree and executes a transformation on the tree (preserving the semantics)
  - Calcite allows to specify custom rules
- Metadata providers: metadata guides the planner to reduce cost and provides info to the rules being applied
  - Information:
    - Overall cost of executing a subexpression in the operator tree
    - Number of rows and data size
    - Max. degree of parallelism that can be executed
    - Information about the plan structure
  - Systems can plug their metadata in different ways
    - Override existing functions
    - Provide custom metadata functions
    - Just provide info about the data
      - Rely on Calcite's default implementation



Apache Calcite query optimizer

- Planner engine: trigger rules until an objective is reached
  - Currently, Calcite provides two:
    - Cost-based: its goal is to reduce the overall expression cost
      - Dynamic programming approach
    - Exhaustive planner: triggers rules exhaustively until it generates an expression that is no longer modified by any rules
      - Useful to quickly execute rules without considering the cost of each expression
  - New engines can be plugged into Calcite
- Materialized views
  - Calcite allows data sources to expose materialized views
  - The optimizer can user these views rewriting incoming queries
  - Two algorithms for this:
    - View substitution
    - Lattices



Apache Calcite Streaming (talk)

- Calcite streaming SQL extension
- Streaming vs relational queries (“pull” vs “push”)
  - Streaming outputs a record whenever one arrives to the stream
    - It is only interested in incoming queries
  - Relational outputs currently existing records and terminates
  - Running a streaming query on a table, or a relational query on a stream gives an error
- STREAM keyword for streaming queries
  - SELECT STREAM * FROM <stream_name>
- Advanced windowing operations
  - Tumbling windows (group by)
    - Monotonic/quasi-monotonic columns required in the group by to detect progress
      - These columns have to be declared in the schema
  - Hopping windows (retain period > emit period)
  - Sliding and cascading windows *
- Views
  - Can be queried as streams or tables
- Joins
  - Table to table
  - Stream to table *
  - Stream to stream *
- Some of these things (*) have not been implemented yet

Apache Calcite drawbacks (or unimplemented features)

- Materialized views
- Indexes (not really needed I guess)
- Certain streaming functionality has not been implemented yet

Stream-table duality paper (Streams and tables: two sides of the same coin)

- It represents the relational notion of table as a stream of successive updates
  - Duality of tables and streams
- Different challenges when defining semantics in the streaming model:
  - Data-related, such as physical vs logical order of events (handled by Chronolog I assume)
  - Nature of streams implies a tradeoff between processing cost, latency and result completeness
  - Handling out-of-order data
- In other models, latency is usually dominated by the characteristics of the data stream rather than the implementation of operators
  - Latency grows linearly with the “maximum lateness” of records, this is, the difference between the timestamp of an out-of-order record and all the previous records
- This paper tries to overcome this proposing the continuously updated model
- This derives in two views on the result of an operator:
  - Static view, the materialization from processing a stream up to an offset
  - Dynamically, a stream of successive updates
  - Also, two ways of programming, processing result updates or continually querying a materialized result

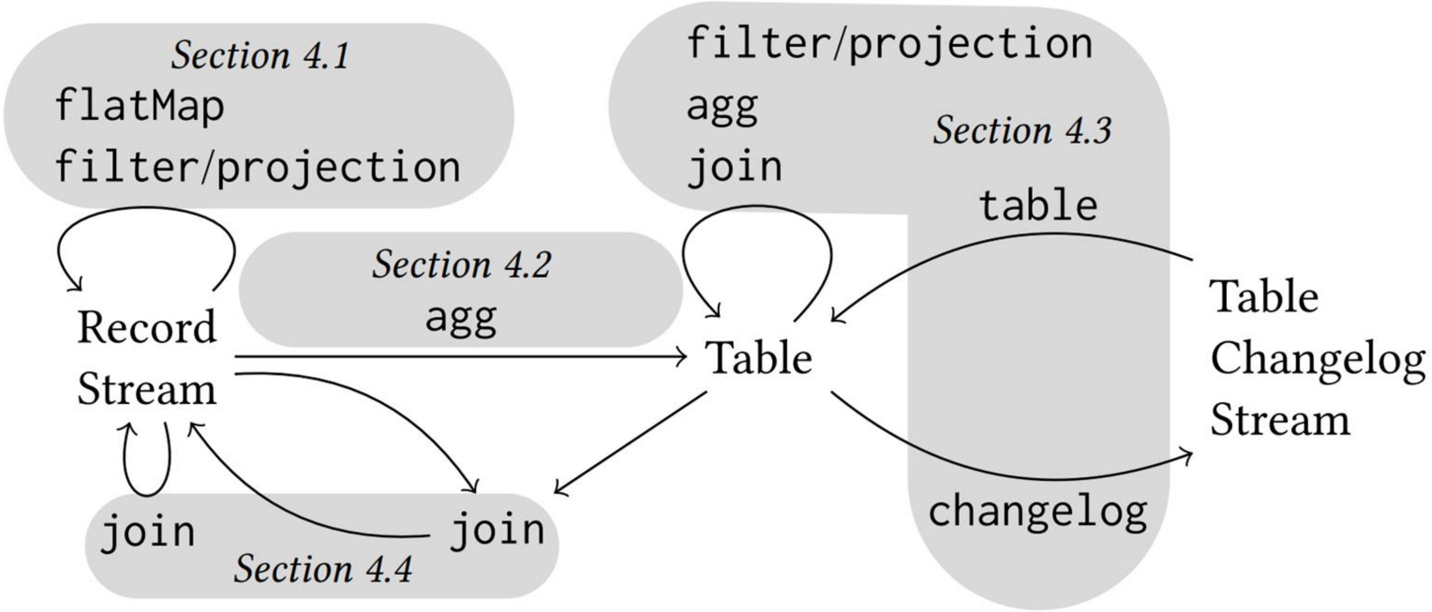
Stream-table duality paper (Streams and tables: two sides of the same coin)

Dual streaming model

- Composed of:
  - Table
    - Static view on the result of an operator, updated as inputs are processed
    - Out-of-order records are handled just like any other record
    - Concept of table as a collection of table versions
  - Table changelog
    - Dynamic view on the result of an operator
    - Consists of records that are updates to a table
    - Applying a table changelog stream to an empty table yields a table with the latest info
    - Each record has a key and the number of unique keys is finite
      - A table materialized from a table changelog stream is finite
  - Stream
    - Records represent facts (and not updates)
    - Each record has a unique key over the whole stream
      - The number of unique keys may not be finite

Stream-table duality paper (Streams and tables: two sides of the same coin)

- Different Streaming operators
- Different output results
- Introduces the tradeoff between always working with materialized views vs sometimes operating over the table changelog stream only



Materialize - Join handling

- Materialize makes heavy use of indexes to optimize joins
- By using primary key indexes and foreign key indexes, it can reduce to zero the need to store intermediate results for a materialized view
- Basically, almost every primary and foreign key needs to be indexed to make total use of their optimizations
- The drawback is that it is needed to maintain a very large collection of indexes
  - Still, according to their documentation, this heavily reduces the amount of information stored for each view
  - And for every new view, indexes are re-used, opposed to working without indexes where every view needs to compute all this information
- At the end, for a fully optimized query, materialize only requires to store twice the number of final records for each materialized view
  - This is, for a query that outputs 200 rows, materialize will store around 400 rows of data
    - This includes the 200 output rows, plus another 200 rows of internal data

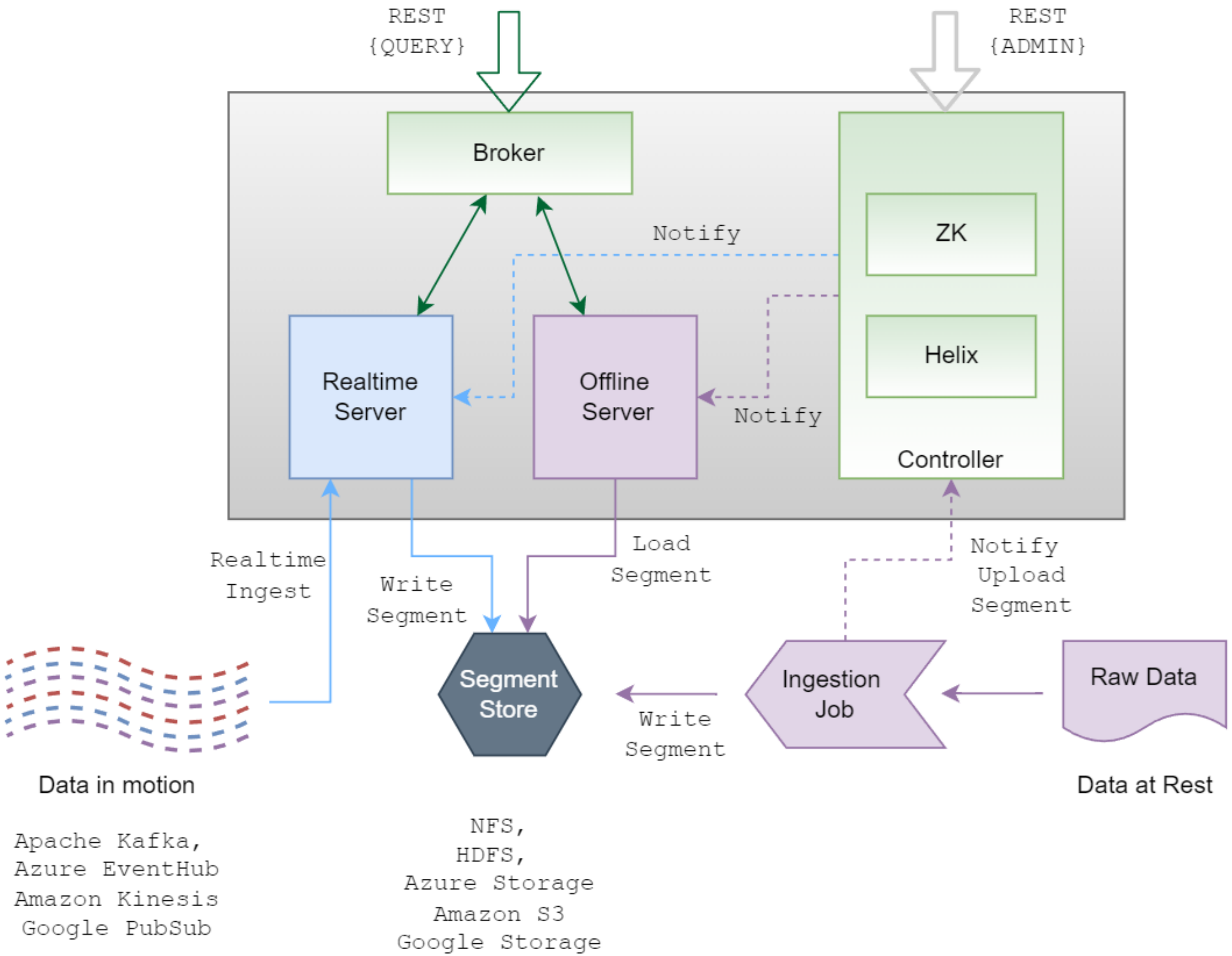
Apache Pinot

- Distributed, real-time, scalable, columnar OLAP data store
- Developed at LinkedIn
- Analytical use cases on immutable append-only data
- Low latency
- For example, used at LinkedIn for customer applications such as “Who viewed my profile”, or to provide content recommendations
- Also used for internal needs such as business analysis dashboards
- Can work with realtime and offline data (lambda architecture)
- PQL query language
- APIs available as well

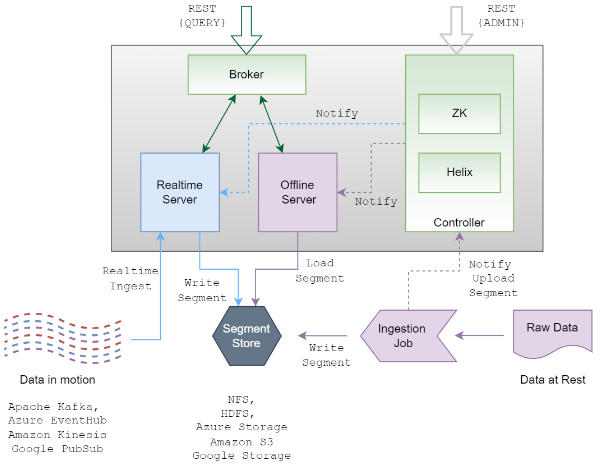
Pinot architecture

- Uses Helix to manage partitions and replicas
  - Helps to optimize query load based on where data is stored in the cluster
  - Establishes three types of nodes
    - Participants - nodes that store the partitions of data
    - Spectators - observe the state of participants and route requests
    - Controllers - observe and manage the state of participants, and coordinates state transitions in the cluster
- Uses Zookeeper to store and update configurations and for distributed coordination
  - Who is the controller of the cluster
  - The list of servers and brokers, their configuration and status
  - List of tables, table schema and configurations
  - Routing tables for every segment, state of the segments

Pinot architecture



Pinot architecture



- Controller
  - Maintains a mapping of segments to servers
  - Lists, adds, and deletes tables and segments
  - For fault tolerance, more than one controller can be
- Broker (spectator)
  - Routes incoming queries to the appropriate server i
  - Collects partial query responses
  - Merges them into a final result and sends it back to the client
  - Can be scaled horizontally
- Server (participant)
  - Store segments
  - Offline servers and real-time servers

Pinot query processing

1. Query is parsed and optimized
2. A routing table for the table involved in the query is selected
3. Servers in the routing table receive the query to be processed, each on a subset of the total segments of the table
4. Servers generate logical and physical query plans
5. After completing all executions, results are gathered, merged and returned to the broker
6. When all results are gathered, partial per-server results are merged together
7. The result is returned to the client

Pinot drawbacks

- PQL is quite limited
  - Supports selection, projection, aggregations, and top-n queries
  - Does not support joins, nested queries, window functions, updates or deletions