Monofeed

High-availability , quality-preserving, quantitative research data infrastructure for cryptocurrency tick-level data

Author: Alejandro Velez

Introduction

Monofeed is a data logging, storage, and retrieval system for crypto security tick-data. It is meant to provide high quality data for quantitative research purposes and maintain high availability in its data pipelines and retrieval services such that quantitative research teams may reliably use the provided data to generate new research datasets consistently and without interruption.

At a high-level, the architectural components are as follows: websocket/application read/write servers, a message queue / transport, a coordinator process (Monofeed), a distributed file system, batch pipelines, core database, and a unified schema/query engine.

Requirements

Functional

- Retrieve and store tick-data fetched from several exchanges.
- Support retrieval from storage for resampling and new dataset generation
- Support for data updates is critical
- Ease of use (ie. ease of querying the system for reads)

Non-functional

- High availability (> 99.99%) is critical
- Data correctness / consistency is critical

Key design decisions

- This system does not prioritize ease of realtime slicing/dicing of the data (joins etc.). It is assumed this is done separately through resampling and aggregation methods defined by Quantitative Researchers. However extensions are proposed to handle such cases. This decision is made to prioritize filter-based retrieval of granular ticks, the specified design priority.
- Latency in updates is heavily deprioritized.
- Data becomes available at a daily cadence.

APIs

GET read(query) : users must be able to read data according to a query. In this design, a query is simply a set of filters (date range, exchange, etc.). Queries specifying joins, or reduce methods(sum, group by, etc.) are not prioritized. However, extensions are proposed to handle such cases.

*PUT update(query) : users should be able to update/correct the data. It is important to note, since the primary purpose of this system is to store tick-level data fetched from an exchange, the likelihood of a user manually performing an update to tick-levels data is very low. So this API is also heavily deprioritized.

Data Model and Schemas

Tick: A tick is the base entity of the system. By default, it contains the following fields, though the system allows for updating such items with *ease*.

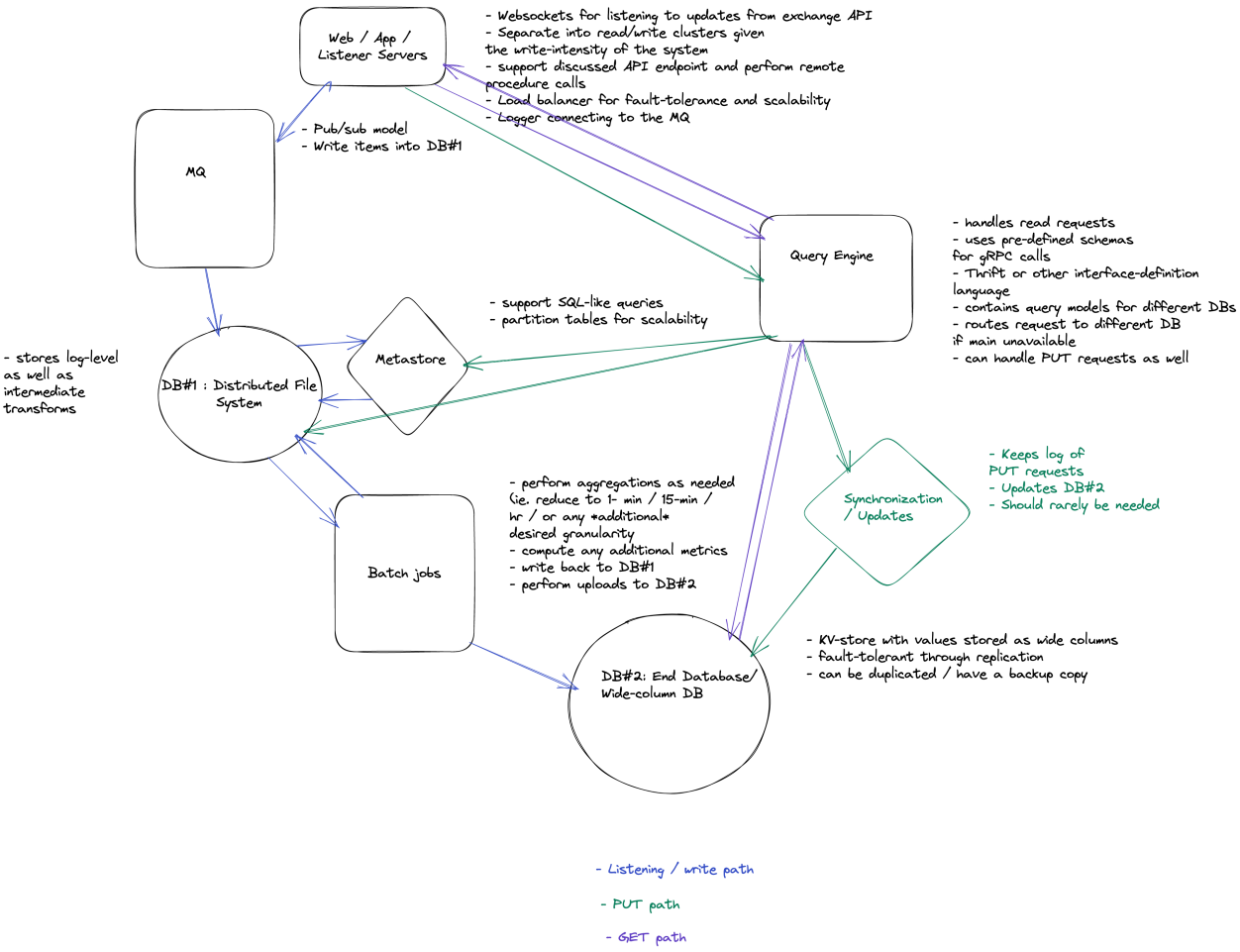
- Exchange [string / char array]: the exchange the tick is fetched from
- Symbol [string / char array]: “ETH-USDT” for example
- Bid [double]: high-precision bid price from the tick
- Ask [double]: high-precision ask price from the tick
- Timestamp [float]: The timestamp pertaining to the tick’s data
- TimeReceived [float]: The timestamp for our receipt of the tick

* it will be easy to update values schema for a tick (ie. Adding a price, volume, or other value), filters will be much harder (ie. Updates to the granularity of or add an additional timestamp you’d like to filter by [not one you simply want to store as a value]).

Query: A query is an abstraction for any read request made to the system. In this implementation, only a filter-based query is explicitly defined (RangeQuery), but this abstraction allows for other queries (joins and other aggregations) as will be addressed in the extensions. Another benefit of this abstraction is the possibility of building a unified metrics layer API for complete consistency among research teams and individual professionals, though this is outside the scope of this design.

- FromDate [string / char array (datetime compatible)]: start date of query
- ToDate [string / char array (datetime compatible)]: end date of query
- Exchange(s): exchange(s) considered in the query
- Symbol(s): symbol(s) to fetch in the query

High-level Design



The entry point of the system is application servers handling listening, get, and put requests. Write and read server clusters are separated given the write-intensity of the system.

Listening process uses a logger and sends logs to the message transporter / queue (MQ), which then uploads logs to a distributed file system via the pub-sub model. Batch jobs perform any intermediate transformations of the logged data (ie. Aggregations to higher-order time units) and also perform uploads to the column-oriented datastore end database.

PUT path should rarely be needed. The request is sent to the query engine which sends updates to the DFS and to the update logger / synchronization service, which can then update DB#2.

GET query process performs a remote procedure call to the query engine utilizing a shared interface definition language. The query engine leverages interface definitions for queries and database requests as well as encoded database query classes/engines to query the required data off of DB#2. If DB#2 is unavailable for any reason, the query engine can route the request to the DFS database / DB#1 for fault-tolerance.

More detailed design details

Web Servers

- Listening is done via web sockets due to it being the default protocol available in most exchanges (because, they support trading requests as well). Due to not needing two-way communication in this design, it is plausible to use a more stable one-way communication mechanism. Long-polling on the exchange REST API can be used given latency is not a big concern, it is possible to have this running as a secondary mechanism in case web sockets fail as well.
- This is a write-intensive system, as such a separate cluster for listening servers is warranted. The write cluster should include load balancing for the same reason.
- Logger service acts as publisher in MQ pub-sub. It can be a daemon process.

MQ

- A message transporter such as Kafka can be used
- Sharding is done to prevent hot spotting and the shard can be done by symbol pair (ie. hash(ETH, USDT) for ETH-USDT symbol) to distribute events from a highly volatile currency as best as possible.
- The complete granular data is uploaded to the DFS. Aggregations of any kind are done in the batch layer.

DFS

- AWS S3 will suffice
- Paired with Hive metastore for ease of data job, table partition, and offline querying.
- JSON is the chosen file format. Others such as parquet offer better latency and compression, but latency is not prioritized here and the compression factor is negligible at this scale. JSON offers human readability.

Batch jobs

- Upload data to DB#2 . This includes lowest-granularity data directly from the DFS logs generated via the MQ as well as any additional transformations generated via data jobs.
- Transformations are written back to DFS as well and jobs can perform metadata updates leveraging Hive Megastore
- A distributed processing tool such as Spark / SparkSQL will suffice

End DB / DB#2

- Column-oriented datastore to support easy schema updates
- KV design supports high availability
- Hash | exchange | symbol | granularityKey | timestamp . Key design will work very well. Given latency is low-priority, hashing is a great prefix choice to assert evenly distributed writes and high availability
- HBase is a great choice given the priority of quality and availability. It has strong consistency guarantee and is highly available. Replication should be enabled and it's wise to include a duplicate DB in case of master failure, though the system handles such a case via failover to the DFS.

Query Engine

- A unified interface definition language schema, via Thrift, allows for the same client request to work across databases and queries.
- Includes engines for all distinct databases
- Can easily support new databases / metric definitions.
- Performs gRPC calls across services.

Additional infrastructure details

- Workflow scheduler needed : use Airflow for workload management
- Container management service via Kubernetes and containerd (Docker ok but being deprecated, albeit at glacier pace)

Limitations and extensions

- There is no support for slicing / dicing / joins of the data in realtime
  - This is assumed to best be left to a different layer. The priority of this system is storage and retrieval of tick-level data at any desired granularity.
  - Adding a batch pipeline dumping into a different end database with stronger realtime analytics capabilities is possible, though pre-aggregations are likely needed. It will be difficult to have this work dumping lowest-level data. Druid can be used here.
  - It is not recommended to add such capabilities to DB#2 via coprocessor architecture as this would disable replication, sacrificing availability.
  - Our unified interface allows for adding a new DB if absolutely needed (MySQL for joins, Druid for RTA) with no change on the user side.

- HBase master as a single point of failure
  - This is addressed via failover to DFS/S3
  - A duplicate cluster is recommended, though not needed under very soft latency guarantees.

- Adding additional filtering values to the row key once implemented is near impossible
  - This really should not happen for tick level data
  - A new table can be formed if absolutely needed
  - Adding new values to the data (new metrics) is very easy thanks to wide column architecture